The Total Cost of (Non) Ownership of a NoSQL Database Cloud Service

Kirill Gavrylyuk, June 28th, 2017
# Contents

Introduction ........................................................................................................................................... 3  

Azure Cosmos DB Overview ................................................................................................................. 4  

Cost Considerations ............................................................................................................................... 6  

  Cost considerations for an on-premises database ............................................................................... 6  

  Cost considerations for running a database using cloud infrastructure ........................................... 6  

  Cost considerations for managed cloud services ............................................................................... 7  

Part I. TCO of a Managed NoSQL cloud service .................................................................................... 7  

  Scenario ........................................................................................................................................... 7  

  Summary of the TCO analysis ........................................................................................................... 8  

Part II. Micro-scenarios .......................................................................................................................... 10  

  One Million Writes per second, sustained load .............................................................................. 11  

  One Million Operations per second, sustained mixed read/write load ............................................ 14  

  One Million Writes per second, regular bursts .............................................................................. 16  

Appendix A. Breakdown of TCO estimates for Part I Scenario. ............................................................ 17  

  Month 1 (Low Usage) ..................................................................................................................... 17  

  Month 2 (Peak Usage) .................................................................................................................... 19  

  Month 3 (medium usage) ............................................................................................................... 21  

Appendix B. Cassandra cluster setup for Part II scenarios. .................................................................. 23
Introduction

Many ongoing and new software development projects today face a choice between deploying a solution on-premises, employing a cloud infrastructure, or using managed cloud services. Many factors are typically involved in this decision. Comparing the total financial cost of owning and operating software in these three deployment configurations is an important factor that requires detailed and careful analysis. The Total Cost of Ownership (TCO) is often the financial metric that is used to estimate and compare direct and indirect costs of a product or a service. While it is challenging to do the right apples-to-apples comparison for the entire solution, comparing the TCO of running off-the-shelf software components (e.g. a database) on-premises vs employing cloud infrastructure, or using a managed cloud database service is more deterministic.

A while ago Amazon published a NoSQL TCO analysis that compared running open source NoSQL databases on-premises on Amazon’s cloud infrastructure, vs using their managed NoSQL cloud service DynamoDB. In the first section of our whitepaper we augment the Amazon study with the TCO for Azure Cosmos DB, the planet scale managed NoSQL cloud database service from Microsoft, for the same scenario. We also update TCO numbers for the cloud hosted OSS NoSQL databases mentioned in Amazon’s whitepaper by using Azure virtual machines equivalent to those used in Amazon paper.

One challenge with the approach taken in Amazon’s whitepaper is the number of assumptions (often not explicitly articulated) made about the cost of running OSS NoSQL database. To start with, the paper does not mention which OSS NoSQL database is being used for comparison. It is difficult to imagine that the TCO of running two very different NoSQL database engines such as Cassandra or MongoDB for the same scenario would be exactly the same. However, we think Amazon’s study maintains its important qualitative merit, this concern notwithstanding.

In the second section of our whitepaper we attempt to address this concern, and provide more precise quantitative comparison for more specific scenarios. We examine two scenarios:

1. Ingesting 1 million records/second
2. A balanced 50/50 read/write workload

We compare the TCO for these micro-scenarios when using the following NoSQL databases:

- Azure Cosmos DB
- Amazon DynamoDB
- Cassandra, a popular OSS NoSQL choice for write-heavy scenarios

In order to run tests with Cassandra, we utilize the open source Cassandra-stress command included in the open source PerfKit Benchmarker.

Our conclusion is consistent with the Amazon’s TCO paper:

1. Managed cloud services can be 5-10x more cost effective. Once all the relevant TCO considerations taken into account, the managed cloud services like Cosmos DB and DynamoDB can be 5-10x more cost effective due to the following reasons:
• **No NoSQL administration dev/ops required.** With Cosmos DB one does not need to employ dev/ops to manage deployments, maintenance, scale, patch and other day to day tasks required with OSS NoSQL cluster hosted on-premises or on cloud infrastructure.

• **Superior elasticity.** Cosmos DB throughput can be scaled up and down on seconds notice, allowing to reduce the cost of ownership during non-peak times. OSS NoSQL clusters deployed on cloud infrastructure offer limited elasticity, and on-premises deployments are not elastic.

• **Economy of scale.** Managed services like Cosmos DB are operating with a really large number of nodes, and are able to pass on savings to the customer.

• **Optimized for Cloud.** Cloud born databases like Cosmos DB, are designed from the ground-up with fine grained multi-tenancy and performance isolation. This allows for optimally placing, executing and balancing thousands of tenants and their workloads across clusters and datacenters extremely efficiently. In contrast, the current generation of OSS NoSQL databases are born to operate on-premises with an entire VM assumed to run a single tenant’s workload. These databases also not designed with to leverage a cloud provider’s infrastructure and hardware. For example, OSS NoSQL database engine is unaware of the differences between a VM going down vs a routine image upgrade, or the fact that premium disk is already 3-way replicated.

2. **Cosmos DB is 10-15% cheaper than DynamoDB for moderate workloads and up to 2-3x cheaper for high volume workloads.** Thanks to predictable performance guaranteed by both offerings, these numbers can be verified by simply comparing the public retail price pages. Cosmos DB offers write optimized low overhead indexing by default making queries more efficient without worrying about secondary indexes. Cosmos DB writes are less expensive, especially for high throughput workloads.

3. **For high volume workloads, Cosmos DB TCO is up to 4 times cheaper than OSS Cassandra on Azure Virtual Machines in all but one studied configuration.** Cassandra is famous for its performance in scenarios involving a sustained high load with write requests, low storage. The TCO of Cosmos DB and Cassandra hosted on D14v2 Azure Virtual Machines is comparable in such configuration. If a mixed set of operations is used, or non-trivial size of data needs to be stored, or the load pattern has bursts, Cosmos DB TCO can be up to 4x lower than that of Cassandra on Azure Virtual Machines.

**Azure Cosmos DB Overview**

Azure Cosmos DB is Microsoft’s globally distributed, multi-model database service. With the click of a button, Azure Cosmos DB enables you to elastically and independently scale throughput and storage across any number of Azure’s geographic regions. It offers throughput, latency, availability, and consistency guarantees with comprehensive [99.99% service level agreements (SLAs)](https://docs.microsoft.com/en-us/azure/cosmos-db/service-level-agreements).

Cosmos DB contains a write optimized, resource governed, schema-agnostic database engine that natively supports multiple data models: key-value, documents, graphs, and columnar. With Cosmos DB, you can take full advantage of many different APIs for accessing data including MongoDB, SQL,
Cassandra, Gremlin (graph), and Azure Table, without dealing with schemas and index management, and the hassles of other on-premises or virtual machine-based cloud options. Cosmos DB offers single digit millisecond latencies at P99, well-defined consistency choices, and limitless scale for all data models and APIs it supports.

Cosmos DB is a truly schema-agnostic – it automatically indexes all the data it ingests without explicitly requiring schema or index management from developers. Internally, it maintains multiple index types (tree-path, hash, range, inverted index and in soon, columnar) to efficiently serve the queries. You can augment your data with business logic as stored procedures, triggers, and UDFs entirely in JavaScript, and have them executed directly inside the database engine, with transactional guarantees spanning multiple documents, entities or graphs.

With Cosmos DB you can write a sustained volume of data and it will be automatically and synchronously indexed to serve consistent queries using a write-optimized, latch-free database engine designed for solid-state drives (SSDs) and low latency access. Default automatic indexing behavior can be customized by a developer for further optimizing performance.

For high-throughput and high-storage workloads Cosmos DB automatically partitions your logical data containers across one or many physical partitions. Your data containers will seamlessly scale out as the quantity of data stored grows and reserved throughput increases.

Cosmos DB data containers can be globally distributed to help you easily build apps with the ability to elastically provision throughput across multiple geographical regions. All of your data is automatically replicated in the regions you specify. Your app continues to work with one logical endpoint, while your data is automatically served from the region closest to your users at low latencies. Execution of the cross region replication operations is free of charge. Cosmos DB allows application to programatically control the failover of write and read regions for testing the end to end availability of the application (beyond just the database). Beyond the ability to simulate the regional failover, Cosmos DB also automatically and transparently performs the failover in the event of regional outage. Cosmos DB also provides transparent multi-homing APIs which does not require the application to be redeployed in the event of a regional failover (either simulated or real).

Regardless of the consistency level, Cosmos DB employs majority write quorums and guarantees that writes are always durably committed on local SSD before ACK’ing the client. Cosmos DB uses flexible read quorums based on the consistency level. The replicas of each partition is spread across 10-20 fault domains and the replica set membership is dynamic to better deal with successive failures. Finally, Cosmos DB’s cross region replication protocol ensures that writes and reads are always served locally (for the three consistency levels applicable in a globally distributed setup).

With Cosmos DB you can store an unlimited amount of data and provision any throughput capacity for a given data container. For each container, you can reserve guaranteed throughput for all supported operations using a uniform measure of throughput called Request Units per second (RUs). For example, a read operation on a 1KB document requires 1 RU. The reserved RUs are independent from the amount of data stored in a container. You can adjust the reserved throughput programatically or via the portal.
at any time after the container is created. Each container is billed hourly based on the amount of data stored (in GBs) and throughput reserved in units of 100 RUs/second.

Cost Considerations

Amazon’s paper summarizes the cost considerations when determining the TCO of a software deployed and operated on-premises, in a cloud infrastructure, or a managed cloud service. Let’s recap what they are.

Cost considerations for an on-premises database

For NoSQL database software deployed on-premises, you will need to acquire the following:

- Physical servers
- Storage disks
- Software licenses (when they are not open source)
- Power and cooling hardware
- Real estate space (or co-located space)
- Administration.

To operate and maintain that same NoSQL storage solution, you will have to consider the cost of the following:

- Intra and inter datacenter redundant storage
- Maintenance of servers and storage arrays
- Overprovisioning of the procured storage
- Cost of redundant storage
- Cost of replacement servers to ensure high-availability
- On-going hardware maintenance of servers, etc.

Redundancy on its own typically increases these costs by at least 3x, depending on your redundancy levels. Furthermore, one quickly realizes that the most significant cost of owning and managing a scalable NoSQL database solution is related to operating and maintaining the software, along with the hardware and infrastructure needed to support it. As your business grows, you will have to add processes in place so that you can quickly add more storage and compute capacity, and this adds more complexity, which further increases your costs.

Cost considerations for running a database using cloud infrastructure

Running NoSQL database software in the cloud significantly reduces infrastructure costs. In the cloud, those costs include the following:

- Instance hours
- GB-month of storage
- I/O requests
- Data transfer
As you add more virtual servers and cloud storage to your solution, your costs increase. You will also have to manage the virtual servers and cloud storage yourself. As the use of your database grows, you will incur additional expense as you manage, operate, and scale the NoSQL database software and its infrastructure environment. This cost comes in the form of hours of time from expert data architects who perform complex scaling techniques.

Cost considerations for managed cloud services
With managed cloud services like Azure Cosmos DB or Amazon DynamoDB, you benefit from the following:

- No direct acquisition costs of database hardware
- No indirect administration costs of managing and scaling your hardware environment

Managed cloud database services handle all this heavy-lifting for you. It frees the IT department from the headaches of provisioning hardware and systems software, setting up and configuring a distributed database cluster, and managing ongoing cluster operations such as patching the OS or NoSQL software. With a few clicks of a mouse in the management portal you can create your own Cosmos DB container and the service is ready to accept API requests from your applications. To scale, you do not need to deploy new infrastructure or perform database sharding. You tell the Cosmos DB service how many requests it needs to be able to handle per second and it automatically spreads your data across enough hardware to provide consistent performance and to protect against down time.

Part I. TCO of a Managed NoSQL cloud service
In this section we will augment the Amazon’s study with corresponding results for Azure Cosmos DB, as well as OSS NoSQL database software running on Azure cloud infrastructure (using Amazon’s approach for cost estimation).

Scenario
Amazon’s paper uses the following scenario. To ensure clean comparison, we kept all parameters and assumptions unchanged.

An organization wishes to leverage NoSQL database technologies for a new application - a new upcoming multi-player social game with characters from a future blockbuster movie. Three different segments in time are considered with the corresponding request rate and storage size:

<table>
<thead>
<tr>
<th></th>
<th>Month 1 (Low)</th>
<th>Month 2 (High)</th>
<th>Month 3 (Medium)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reads (per second)</strong></td>
<td>50</td>
<td>3500 (peak)</td>
<td>1200 (peak)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1200 (off-peak)</td>
<td>700 (off peak)</td>
</tr>
<tr>
<td><strong>Writes (per second)</strong></td>
<td>25</td>
<td>1500 (peak)</td>
<td>800 (peak)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800 (off-peak)</td>
<td>300 (off-peak)</td>
</tr>
</tbody>
</table>
Month 1: In the first month, since the game was launched with little marketing and the movie was still not released, the game did not require more than 50 reads per second and 25 writes per second. At the end of the month, the game accumulated approximately 200 GB of data.

Month 2: In the second month, the movie was released and the game gained popularity and experienced a large spike in traffic with thousands of users accessing the game simultaneously. Users were consistently accessing the game at the rate of 5,000 requests per second during peak times and 2,000 requests per second during off-peak times. Data usage increased quickly to 900 GB. Data egress increased to 1.5TB.

Month 3 and beyond: In the third month, the movie buzz faded. As a result, the traffic subsided, and the demand decreased for the game. Reads and writes dropped to 2,000 per second during peak hours and 1000 per second during off-peak hours. At the end of the month, the game accumulated approximately 1,200 GB of data. Data egress reduced to 600GB. In subsequent months the usage remained the same.

Summary of the TCO analysis
Here is the total cost of ownership for the above scenario for each month in each of the following implementation options:

- **NoSQL on Premises.** OSS NoSQL database (such as Cassandra or MongoDB) deployed on premises, using estimation methodology described in Amazon’s paper, updated to today’s prices based on Amazon’s TCO calculator.
- **NoSQL on Azure VMs.** OSS NoSQL database deployed in Azure cloud infrastructure, based on Azure public pricing for Linux Virtual Machines and Premium Disks.
- **Amazon DynamoDB.** Based on Amazon public pricing.
- **Azure Cosmos DB.** Based on Azure public pricing.

<table>
<thead>
<tr>
<th>Data accumulated (GB)</th>
<th>200</th>
<th>900</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Egress (GB)</td>
<td>200</td>
<td>1500</td>
<td>600</td>
</tr>
</tbody>
</table>
To ensure a true comparison, we kept the TCO methodology unchanged as described in Amazon’s paper and simply extended the analysis to include Azure Cosmos DB and a NoSQL database running on Azure cloud infrastructure. We also used the current prices for Amazon’s DynamoDB. For on-premises deployment we used Amazon’s TCO calculator to use the current hardware prices, and yet ensure consistency with the original calculations made in Amazon’s paper.

Relative results are consistent with those published in Amazon’s paper.

- The TCO for Cosmos DB and DynamoDB for each month is comparable (Cosmos DB is ~10-15% cheaper).
- The TCO for OSS NoSQL databases running on Azure VMs is up to 5x more expensive than Cosmos DB.
- The TCO for NoSQL on premises can be up to 15x more expensive than managed cloud services, especially during Month 3 where on-premises hardware utilization drops.

Here are few factors driving the higher cost for on-premises and cloud infrastructure based solutions:

1) On-premises deployment requires non-trivial hardware acquisition, maintenance (power, cooling, etc.) and operations investment. Hardware cost is amortized here over 3 years.

2) On-premises deployment does not offer elasticity and requires over-provisioning. On month 3 and beyond, the hardware utilization is dropped, but the money is already spent.

3) Cloud infrastructure based solutions offer limited and coarse-grained options for adjusting throughput capacity. While interviewing customers and teams in Microsoft using OSS NoSQL solutions, we observed optimistic configurations often obtained during the first testing phase, followed by overprovisioning VMs to account for unforeseen obstacles, and ensure availability when the system is in production.

4) Both on-premises and cloud infrastructure based solutions require database administration.

5) Cloud infrastructure based solutions do not take full advantage of the cloud capabilities, since OSS NoSQL software is not optimized for specific cloud providers. For example, OSS NoSQL software is unaware of the differences between a node going down vs a routine image upgrade, or the fact that Premium disk is already 3-way replicated.
Appendix A details the estimation for a NoSQL database on Azure VMs, as well as Cosmos DB options for each of the months. For on-premises NoSQL database deployment options we followed the estimation methodology from the original Amazon paper as-is. Hardware prices have come down since the paper was published, so the table above uses the current, lower hardware prices, using the Amazon’s TCO calculator, just like the original Amazon paper did.

For the Amazon DynamoDB option, public pricing was used. Strong consistency reads are assumed for both DynamoDB and Azure Cosmos DB estimates.

Part II. Micro-scenarios

In this section we explore the publicly documented micro-scenarios representing common NoSQL usage patterns. Our goal is to understand and compare TCO for various common use cases. To avoid any possible bias, we used a widely recognized open sourced benchmark, “one million operations per second” not affiliated in any way with Microsoft:

1. **Writes only, with storage.** One million writes per second, a popular test for Cassandra deployments, using small <1K size records. We included storage required to sustain 12h time-to-live for the written data.
2. **Writes only, low storage.** A variation of #1 assuming storage size is small either due to a very low TTL, or due to writes being upserts.
3. **Reads and writes, eventual consistency.** One million operations per second with 50/50 read/write split, with eventually consistent reads.
4. **Reads and writes, strong consistency.** One million operations per second with 50/50 read/write split, with strongly consistent reads.
5. **Write bursts.** One million writes per second, bursty workload with peaks every 4 hours.

For OSS Cassandra we used the Open Source PerfKitBenchMarker stress test using write operation as a stress command. We ran the tests with OSS Cassandra 2.1 on Azure Virtual Machines D14v2. For Cosmos DB and DynamoDB we relied on the provisioned throughput guarantee and used public prices for the required provisioned throughput.

For each test we calculate TCO including compute cost, network bandwidth as appropriate, and Cassandra administration. In all tests we let Cassandra run for 24h prior to the test to ensure we have reached compaction. We have done a basic optimization of the Cassandra cluster installation on Azure to the best of our ability using both public recommendations, as well as excellent notes from Al Tobey. We are confident you can do better, and we’d love to improve, please contact kirillg@microsoft.com if you are interested in helping to optimize this further.

We were not able to achieve 1 million operations per second with MongoDB, and were not able to find anyone else publicly able to achieve this, so we are not including results for MongoDB here at this time.

Our main learnings include:
• As expected, Cassandra does very well in sustained write-heavy scenario with low storage. While the compute cost for this scenario with Cassandra is low, the overall TCO of running Cassandra for these scenarios is comparable or higher than that of Cosmos DB, driven by dev/ops overhead. We see this overhead sometimes ignored during initial project stages, only to cause headaches later on.

• When the workload has a mix of read and write requests, Cosmos DB TCO tends to be 50% lower than that of Cassandra on Azure VMs thanks to lower cost for Cosmos DB reads and queries.

• Cassandra-based system costs often include an additional storage component if the local node’s disk drive is not sufficient (either because data needs to be retained beyond few hours, or disaster recovery is a requirement). This extra storage cost with Cassandra is 4x higher than that with Cosmos DB or DynamoDB, driven in part by Cassandra’s unawareness of the underlying cloud platform.

• For bursty, uneven workloads, managed database services like Cosmos DB and DynamoDB offer better elasticity compare to non-managed NoSQL solutions running on IaaS, allowing for up to 10x savings in TCO.

• Due to optimized low-overhead full document indexing by default, Cosmos DB is much more efficient for reads and queries than Cassandra and DynamoDB, and does not require worrying about secondary indexes.

• For high volume scenarios, Cosmos DB’s TCO is 2x-3x lower than that of DynamoDB, thanks to low-overhead indexes and an optimized per-write cost for high volume write scenarios.

Below chart is the summary of the hourly TCO for each of the five scenarios considered.

![Hourly TCO, one million operations/sec](chart)

One Million Writes per second, sustained load
This test has been repeated on several occasions with Cassandra, most notably by Netflix using Amazon instances, and Google using Google compute.
The test represents a common situation in IoT or cloud telemetry where a sustained stream of 1M records per second is received from a large number of devices 24x7. Record size is 1K. Typically raw data like that is not kept for too long, or gets immediately aggregated and archived to cold storage. So we assume the data has ~12h TTL, which means the system needs to maintain ~43TB of storage at any given time.

Summary
This test is traditionally used to advertise Cassandra, so we expected it to perform well. While the cost of compute needed for Cassandra is lower than that of provisioned throughput with Cosmos DB (or DynamoDB), the database administration becomes a significant component of Cassandra TCO that is often ignored during early cycle evaluations. As a result, Cosmos DB quorum writes have 20% lower TCO than Cassandra quorum writes (and more than twice lower than DynamoDB).

Another component that drives Cassandra’s overall TCO higher is storage. Since our storage needs exceed what is available on VM’s local drive, we’re using Premium SSD-backed disks as per recommendation, that are already 3-way replicated. Unfortunately, there is no way to instruct Cassandra to take advantage of that additional redundancy. As a result, incremental storage costs for Cassandra clusters on Azure come to ~ $1.08/GB ($0.135 per GB, on each of the four replicas, buffered 50% for compaction), which is 4x the cost of storage with Cosmos DB or DynamoDB. This is an example where OSS solutions are not usually optimized for a specific cloud platform.

Since both Cosmos DB and DynamoDB perform reliable quorum writes, we tested with Cassandra write consistency CL=QUORUM. For completeness, we also performed a test with Cassandra write consistency CL set to ONE, which means a write is acknowledged as soon as it is written to at least one replica’s log/memtable. A single node failure with this configuration may lead to (temporary) data loss. If occasional temporary data loss is not a concern, one can achieve 7% cost savings with Cassandra CL=ONE.

We also included a variation of the results with storage cost excluded (simulating a very low storage scenario). In the case of Cassandra, this means the local drive on the nodes can be used as a data store. We did not include this variation for Cassandra CL=ONE, given such system would be susceptible to permanent data loss when a single node fails, not appropriate for the scenarios.
**Breakdown**

**Cassandra CL=QUORUM.** We were able to achieve 1M writes per second with CL=QUORUM using 120 D14v2 nodes. To maintain ~43TB of storage, we needed to attach 3 P30 premium disks (3TB) on each node, allowing 50% for compaction and accounting for replica count=4. Managing Cassandra cluster carries a dev/ops administration cost. We assume 1 full time employee (FTE) is needed for every 100 nodes @$120K annual salary. Hourly TCO for this configuration was $316, including $178 for compute, $68/h for premium storage, and $71/h for administration.

**Cassandra CL=ONE.** With the Cassandra configuration discussed above and a consistency setting CL=ONE, we were able to achieve 1M requests per second with 84 D14v2 nodes. To maintain ~43TB of storage, we needed to attach 4 P30 premium disks to each node, allowing 50% for compaction and accounting for a replica count of 4. Hourly TCO comes to $237 including $124 for compute, $63 for Premium Storage, and $49 for administration.

**Cosmos DB.** Cosmos DB offers reduced RU charges for create, update, and delete (CUD) operations for high volume customers. For the test document we used, our writes carry a charge of 3 Request Units resulting in an hourly TCO for Cosmos DB of $240 for provisioned throughput and $15 for storage.
**DynamoDB.** With DynamoDB we need to provision 1M write units ($0.00065/h per unit) and pay for 43TB of storage ($0.25/GB/month). The hourly TCO for DynamoDB is: $650 for provisioned throughput, and $15 for storage.

**One Million Operations per second, sustained mixed read/write load**
This scenario corresponds to device profile, user profile, or session store workloads, with a balanced 50/50 mix of writes and reads. We evaluated two variations, with strong and weak read consistency. Strong consistency is commonly the default starting point for such workloads, as customers are more comfortable thinking about the systems this way. Data loss in these scenarios is typically unacceptable, so we kept the Cassandra write consistency setting at CL=QUORUM. We assume the same size of data needs to be stored, 43TB, although most real examples for this profile that we examined had larger storage to throughput ratio.

**Summary**
We needed the same number of Cassandra nodes to sustain a mixed workload with strong consistency reads as with a quorum write-only workload, which is consistent with the findings in the Netflix report for mixed read/write workload. In other words, we observe that Cassandra quorum reads have a similar (or slightly higher) cost as quorum writes. On the other hand, reads are much cheaper than writes with both Cosmos DB and DynamoDB.

Just like with the previous test, while the compute cost of Cassandra was slightly lower than that of provisioned throughput for Cosmos DB, the overall TCO of Cassandra is ~50% higher than Cosmos DB, driven by extra administration and storage overhead.

Cosmos DB offers five intuitive consistency settings: Eventual, Prefix, Session, Bounded Staleness, and Strong. While few venture to use eventual consistency, we often see customers using Session consistency for the scenario discussed. It offers each client a consistent view of the database, i.e. each client will read their latest writes, and will get all other updates in-order, guaranteed. For the scenario discussed this is often as good as strong consistency, while the cost of reads with Session consistency is the same as eventual consistency. This additional optimization allows us to bring the overall TCO further down by 20%.
Breakdown: Strong consistency

**Cassandra.** We were able to achieve simultaneous 500K reads & 500K writes/updates per second with CL=QUORUM using 120 D14v2 nodes. To maintain ~43TB of storage, we needed to attach 3 P30 premium disks (3TB) to each node, allowing 50% for compaction and accounting for a replica count of 4. Managing Cassandra cluster carries dev/ops administration costs. Just like before, we assume 1 FTE is needed for every 100 nodes at a $120K annual salary. Hourly TCO for Cassandra in this scenario is $316 including $178/h for compute, $68/h for Premium Storage, and $71/h for administration.

**Cosmos DB.** Cosmos DB strong consistency reads per second cost 2RUs or $0.00016/h, while writes cost 3RUs just like in the previous test. Hourly TCO for Cosmos DB therefore is $215 including $200 for provisioned throughput and $15 for storage.

**DynamoDB.** DynamoDB strong consistency reads cost $0.00013/h per hour. The hourly TCO for DynamoDB therefore is $405, including $390 for provisioned throughput, and $15 for storage.
Breakdown: Eventual consistency

**Cassandra.** We were able to achieve simultaneous 500K reads & 500K writes/updates per second with read consistency setting CL=ONE using 100 D14v2 nodes. To maintain ~43TB of storage, we needed to attach 4 P30 premium disks (4TB) to each node, allowing 50% for compaction and accounting for replica count=4. Managing Cassandra cluster carries dev/ops administration cost. Just like before we assume 1 FTE is needed for every 100 nodes @$120K annual salary. Hourly TCO for Cassandra in this scenario is $282 including $148/h for compute, $75/h for Premium Storage, and $59/h for administration.

**Cosmos DB.** Cosmos DB eventual consistency reads per second cost 1RUs or $0.00008/h, while writes cost 3RUs just like in the previous test. Hourly TCO for Cosmos DB therefore is $175 including $160 for provisioned throughput and $15 for storage.

**DynamoDB.** DynamoDB eventual consistency reads cost $0.000065/h per hour. The hourly TCO for DynamoDB therefore is $372, including $358 for provisioned throughput, and $15 for storage.

One Million Writes per second, regular bursts

A common variation of the write-intensive scenario we find in IoT scenarios has the traffic coming from devices in predictable, regular bursts. Consider the scenario where every 4 hours we receive a 15min long burst of data from devices at up to 1M writes/second. Accurate just-in-time anomaly alert reporting requires us to keep the end-to-end latency low and eliminates the possibility to back off the traffic via queue. This scenario highlights the benefit of PaaS solutions. Both Cosmos DB and DynamoDB can be scaled up and down fast without any side effects on the system throughput, however it would be impractical to scale up and down Cassandra cluster with such frequency.

In this scenario, the TCO comparison changes significantly, as we only pay for every 4th hour for the provisioned throughput for Cosmos DB/DynamoDB, while continue to maintain the same size of Cassandra cluster. Assuming the data needs to be stored for 24h, the amount of data needed to be stored has reduced to 5.4TB, allowing us to use local SSD drive on Cassandra nodes. Below is the comparison chart for the amortized hourly TCO for each system.

<table>
<thead>
<tr>
<th>Cost Items</th>
<th>Cassandra, CL=QUORUM</th>
<th>Cassandra, CL=ONE</th>
<th>Amazon DynamoDB</th>
<th>Azure Cosmos DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td></td>
<td>$163</td>
<td>$60</td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td>$178</td>
<td>$124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage</td>
<td></td>
<td></td>
<td>$2</td>
<td>$2</td>
</tr>
<tr>
<td>NoSQL Administration</td>
<td>$71</td>
<td>$49</td>
<td>$0</td>
<td>$0</td>
</tr>
<tr>
<td>TCO</td>
<td>$248</td>
<td>$174</td>
<td>$165</td>
<td>$62</td>
</tr>
</tbody>
</table>
Appendix A. Breakdown of TCO estimates for Part I Scenario.

Month 1 (Low Usage)
During the first month, since the game was launched with little marketing and the movie was not released yet, the game did not require more than 50 reads per second and 25 writes per second. At the end of the month, the game accumulated approximately 200 GB of data with cumulative egress of 200GB during the month.

<table>
<thead>
<tr>
<th>Cost Items (Amortized)</th>
<th>NoSQL on premises</th>
<th>NoSQL on Azure VMs</th>
<th>Amazon DynamoDB</th>
<th>Azure Cosmos DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Costs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server Hardware</td>
<td>$53.83</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Network Hardware</td>
<td>$10.77</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Hardware Maintenance</td>
<td>$24.23</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Power and Cooling</td>
<td>$25.92</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Data Center / Co-located Space</td>
<td>$53.57</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Hardware Administration</td>
<td>$400.00</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Read Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$4.68</td>
<td>$3.00</td>
</tr>
<tr>
<td>Write Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$11.70</td>
<td>$10.50</td>
</tr>
<tr>
<td>Cloud Compute Resources</td>
<td>$0.00</td>
<td>$458.00</td>
<td>$16.38</td>
<td>$24.00</td>
</tr>
<tr>
<td>Total Compute Costs</td>
<td>$568.32</td>
<td>$458.00</td>
<td>$16.38</td>
<td>$24.00</td>
</tr>
<tr>
<td>Redundancy Costs</td>
<td>$1,136.63</td>
<td>$916.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
The Total Cost of (Non) Ownership of a NoSQL Database Cloud Service

<table>
<thead>
<tr>
<th></th>
<th>$300.00</th>
<th>$219.66</th>
<th>$50.00</th>
<th>$50.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage (3-way rep.)</td>
<td>$300.00</td>
<td>$219.66</td>
<td>$50.00</td>
<td>$50.00</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>$16.00</td>
<td>$17.40</td>
<td>$24.00</td>
<td>$17.40</td>
</tr>
<tr>
<td>NoSQL Administration</td>
<td>$400.00</td>
<td>$400.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>TCO</td>
<td>$2,420.95</td>
<td>$2,011.06</td>
<td>$90.38</td>
<td>$91.40</td>
</tr>
</tbody>
</table>

NoSQL database on Azure Linux Virtual Machines and Premium Disks

- **Compute cost:** $458
  A single MongoDB shard/logical Cassandra node is sufficient for the load. We used Linux D4 VM, 8 core/28GB RAM, running in the East US Azure region. Additional redundancy (two instances) costs were due to a replica count of 3. Note that to achieve higher than 99.9 SLA, the system needs higher replica counts to account for regular infrastructure maintenance.

- **Additional redundancy cost** $916 assumes replica count 3.
  Note that to achieve higher than 99.9 SLA, the system needs a higher replica count to account for regular infrastructure maintenance.

- **Storage cost:** $220
  Premium P20 Azure Disks were used, one per compute instance. We could use a local disk (at zero additional storage cost), however this would tie the storage and compute scalability going forward, as well leave us without disaster recovery plan.

- **Data transfer cost:** $17.4 at $0.087/GB rate in East US.

- **NoSQL administration costs:** $400.
  We used Amazon’s estimation approach as is (1 dev/ops can manage 25 logical server configurations, at 120K USD annual salary). During our interviews with teams using Mongo or Cassandra, an estimate of 1 dev/ops per 60 Mongo shards/logical Cassandra nodes has come up, leading to slightly higher cost. Real costs for small to medium size projects can be higher as administration is performed by a higher paid employee, part time.

The total cost of running NoSQL database on Azure Virtual Machines during the first month is **$2,420.95**.

Azure Cosmos DB

- **Provisioned throughput:** $24 (minimum available)
  Azure Cosmos DB uses RequestUnits/second (RUs) as a measure of provisioned throughput. Each operation takes certain amount of RUs to complete. To sustain 50 reads per second of 1K documents, 50 RUs are needed. To sustain 25 writes per second of 1K documents with default indexing policy, ~175RUs are required. The total is well below the minimum 400 RUs that can be provisioned per container at $24/month.

- **Storage:** $50/month, at $0.25 per GB flat rate.
• Data transfer cost: $17.40, at $0.087/GB rate in East US

The total cost of using Azure Cosmos DB during the first month is **$91.40**.

**Month 2 (Peak Usage)**

In the second month, the movie was released and the game gained popularity and experienced a large spike in traffic with thousands of users accessing the game simultaneously. Users were consistently accessing the game at the rate of 2500 reads and 1500 writes per second during peak hours (75% of the time) and 2,000 reads and 1,200 writes per second during off-peak hours (25% of the time). Data usage increased quickly to 900 GB. Data egress grew to 1.5TB.

<table>
<thead>
<tr>
<th>Cost Items (Amortized)</th>
<th>NoSQL on premises</th>
<th>NoSQL on Azure VMs</th>
<th>Amazon DynamoDB</th>
<th>Azure Cosmos DB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Costs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server Hardware</td>
<td>$269.17</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Network Hardware</td>
<td>$53.83</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Hardware Maintenance</td>
<td>$121.13</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Power and Cooling</td>
<td>$129.60</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Data Center / Co-located Space</td>
<td>$267.86</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Hardware Administration</td>
<td>$2,000.00</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Peak Read Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$327.60</td>
<td>$210.00</td>
</tr>
<tr>
<td>Peak Write Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$702.00</td>
<td>$630.00</td>
</tr>
<tr>
<td>Off-peak Read Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$112.32</td>
<td>$72.00</td>
</tr>
<tr>
<td>Off-peak Write Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$374.40</td>
<td>$336.00</td>
</tr>
<tr>
<td>Cloud Compute Resources</td>
<td>$0.00</td>
<td>$1,259.50</td>
<td>$893.88</td>
<td>$732.00</td>
</tr>
<tr>
<td><strong>Total Compute Costs</strong></td>
<td>$2,841.58</td>
<td>$1,259.50</td>
<td>$893.88</td>
<td>$732.00</td>
</tr>
<tr>
<td>Redundancy Costs</td>
<td>$5,683.16</td>
<td>$2,519.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Storage (3-way replication)</td>
<td>$1,350.00</td>
<td>$658.98</td>
<td>$225.00</td>
<td>$225.00</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>$116.58</td>
<td>$130.50</td>
<td>$135.00</td>
<td>$130.50</td>
</tr>
<tr>
<td>NoSQL Administration</td>
<td>$400.00</td>
<td>$400.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
The Total Cost of (Non) Ownership of a NoSQL Database Cloud Service

| TCO Month 2 | $10,391.33 | $4,967.98 | $1,253.88 | $1,087.50 |

NoSQL database on Azure IaaS / Premium Disks

- **Compute cost:** $1260
  Three MongoDB shard/logical Cassandra nodes are sufficient for the peak load. We used Linux D4 VM, 8 core/28GB RAM, running in the East US Azure region, similar to what was used in Amazon’s paper. The Amazon paper suggests scaling down the NoSQL cluster from 3 nodes to 2 during the off-peak hours (6h daily). The estimate reflects this. However, our discussions with Mongo and Cassandra customers suggests it is not practical to scale down/up daily as re-partitioning the data can quickly become a costly operation affecting solution’s capacity (and not accounted for in Amazon’s methodology). It is likely that a steady three node deployment would be used in practice for the scenario. So a more realistic estimate would be $1374.

- **Additional redundancy cost:** $2519 assumes a replica count of 3.
  Note that to achieve higher than 99.9 SLA, the system needs a higher replica count to account for regular infrastructure maintenance. Also, the Amazon paper suggests scaling down the NoSQL cluster from 3 nodes to 2 during the off-peak hours (6h daily). The estimate reflects this. However, our discussions with Mongo and Cassandra customers suggests it is not practical to scale down/up daily as re-partitioning the data can quickly become a costly operation affecting solution’s capacity (and not accounted for in Amazon’s methodology). As mentioned above, a more realistic estimate would be $2748.

- **Storage cost:** $659
  Premium P20 Azure Disks were used, one per compute instance. We could use the local disk (at zero additional storage cost), however this would tie the storage and compute scalability going forward, as well leave us without a disaster recovery plan.

- **Data transfer cost:** $131 for 1.5TB egress at $0.087/GB rate in East US.

- **NoSQL administration costs:** $400.
  Same as calculated above in Month 1.

The total cost of running NoSQL database on Azure Virtual Machines during the second month is $4,967.98.

**Azure Cosmos DB**

- **Provisioned throughput:** $732
  Peak workload: $840 for 3500 RUs to sustain 3500 reads/sec and 10500 RUs to sustain 1500 writes/sec at $0.06 per month. Off-peak workload: $408 for 1200 RUs to sustain 1200 reads/sec and 5600 RUs to sustain 800 writes/sec at $0.06 per month. Total cost of provisioned throughput is a weighted average reflecting 18h of peak and 6h of off-peak traffic daily.
  Unlike with NoSQL on VMs, scaling up and down daily with managed services like Cosmos DB (or DynamoDB) is straightforward and at a reasonable practice.
Note: Amazon's paper had a calculation error in the DynamoDB cost for provisioned throughput. Instead of weighted average, the paper just added the cost for peak and off-peak workload as if they were running in parallel. We corrected this in our updated DynamoDB estimates.

- **Storage**: $225/month for 900GB at $0.25 per GB flat rate.
- **Data transfer cost**: $130.5 for 1500GB at $0.087/GB rate in East US

The total cost of using Azure Cosmos DB during the second month is **$1,087.50**.

**Month 3 (medium usage)**

In the third month, the movie buzz faded. As a result, the traffic subsided, and the demand decreased for the game. Reads and writes dropped to 1200 per second and 800 per second respectively during peak hours and 700 per second and 300 per second respectively during off-peak hours. At the end of the month, the game accumulated approximately 1,200 GB of data. Data egress totaled 600GB.

### Cost Items (Amortized)

<table>
<thead>
<tr>
<th>Cost Items (Amortized)</th>
<th>NoSQL on premises</th>
<th>NoSQL on Azure VMs</th>
<th>Amazon DynamoDB</th>
<th>Azure Cosmos DB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute Costs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server Hardware</td>
<td>$269.17</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Network Hardware</td>
<td>$53.83</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Hardware Maintenance</td>
<td>$121.13</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Power and Cooling</td>
<td>$129.60</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Data Center / Co-located Space</td>
<td>$267.86</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Hardware Administration</td>
<td>$2,000.00</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Peak Read Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$112.32</td>
<td>$72.00</td>
</tr>
<tr>
<td>Peak Write Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$374.40</td>
<td>$336.00</td>
</tr>
<tr>
<td>Off-peak Read Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$65.52</td>
<td>$42.00</td>
</tr>
<tr>
<td>Off-peak Write Cost</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$140.40</td>
<td>$126.00</td>
</tr>
<tr>
<td>Peak workload</td>
<td>$0.00</td>
<td>$916.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Off-peak workload</td>
<td>$0.00</td>
<td>$458.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Cloud Compute Resources</td>
<td>$0.00</td>
<td>$801.50</td>
<td>$416.52</td>
<td>$348.00</td>
</tr>
<tr>
<td><strong>Total Compute Costs</strong></td>
<td>$2,841.58</td>
<td>$801.50</td>
<td>$416.52</td>
<td>$348.00</td>
</tr>
<tr>
<td><strong>Redundancy Costs</strong></td>
<td>$5,683.16</td>
<td>$1,603.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td><strong>Storage (3-way replication)</strong></td>
<td>$1,800.00</td>
<td>$658.98</td>
<td>$300.00</td>
<td>$300.00</td>
</tr>
<tr>
<td><strong>Data Transfer</strong></td>
<td>$46.60</td>
<td>$52.20</td>
<td>$54.00</td>
<td>$52.20</td>
</tr>
<tr>
<td><strong>NoSQL Administration</strong></td>
<td>$400.00</td>
<td>$400.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td><strong>TCO Month 3</strong></td>
<td>$10,771.35</td>
<td>$3,515.68</td>
<td>$770.52</td>
<td>$700.20</td>
</tr>
</tbody>
</table>

**NoSQL database on Azure VMs / Premium Disks**

- **Compute cost**: $802
Two MongoDB shard/logical Cassandra nodes are used for the peak load, one for the off-peak load. Same instance size is used as in month 1 & 2. Just like for month 2, the estimate reflects scaling up and down between peak and off-peak hours, following Amazon’s paper assumptions. A more realistic assumption would be to keep the same two instances during off-peak hours at $916 monthly cost.

- Additional redundancy cost of $1603 assumes a replica count of 3, following Amazon’s paper assumptions. A more realistic estimate would be $1832 for keeping the same number of instances throughout peak and off-peak times.

- Storage cost: $659, same as during Month 2. Premium P20 Azure Disks are used, one per compute instance.

- Data transfer cost: $52 for 600GB egress at $0.087/GB rate in East US.

- NoSQL administration costs: $400. Same as calculated above in Month 1 & 2.

The total cost of running NoSQL database on Azure Virtual Machines during the second month is $3515.68.

Azure Cosmos DB

- Provisioned throughput: $348
  Peak workload: $408 for 1200 RUs to sustain 1200 reads/sec and 5600 RUs to sustain 800 writes/sec at $0.06 per RUs per month. Off-peak workload: $168 for 700 RUs to sustain 700 reads/sec and 2100 RUs to sustain 300 writes/sec at $0.06 per RUs per month. Total cost of provisioned throughput is the weighted average reflecting 18h of peak and 6h off-peak traffic daily. Unlike with NoSQL on VMs, scaling up and down daily with managed services like Cosmos DB (or DynamoDB) is straightforward and at a reasonable practice.

  Note: Amazon’s paper had a calculation error in DynamoDB cost for provisioned throughput for Month 2 and Month 3. Instead of weighted average, the paper just added the cost for peak and off-peak workload as if they were running in parallel. We corrected this in our updated DynamoDB estimates.

- Storage: $300/month for 1200GB at $0.25 per GB flat rate.

- Data transfer cost: $54 for 600GB at $0.087/GB rate in East US

The total cost of using Azure Cosmos DB during the second month is $700.20.
Appendix B. Cassandra cluster setup for Part II scenarios.

We used the Open Source PerfKitBenchMarker that provides convenient setup scripts to setup Cassandra and execute cassandra_stress/collect dstat and other metrics. Our plan is to share the scripts with subsequent updates of this paper, as we move the automation to use Azure Resource Model templates.

We used 16-core D14v2 for Cassandra nodes as recommended in the DataStax paper, as well as observed in our own tests as compare to D4v2 and D5v2. We also used D14v2 for the client nodes as it offers a much better network, but didn't include the client in our TCO comparison.

Clients and servers were in the same datacenter, cross-node and client-server communication is over internal IP addresses. We used a replica count of 4, which is the theoretical minimum required to match Cosmos DB SLA guarantees of 99.99%.

We used a heap size of 30GB and 8GB for HEAPSIZENEW. All concurrent_xxx config knobs were set to 128. Both changes made significant improvements. We used JDK 1.7, and stayed with CMS GC. We toggled with other knobs mentioned in Al's article but didn’t get a lot more improvement.

With consistency setting CL=ONE we were able to get to 70-80% CPU utilization on D14v2 nodes in all tests. With CL=QUORUM we peaked at 50-60 CPU utilization in all tests. The client was not a bottleneck, and neither was disk. All signs pointed to network I/O, but more investigation is required to understand what else can be tuned, since in/out traffic on a given worker node never exceeded 1.5gbps and iperf consistently allows for 3.2 gbps between D14v2 nodes. Other than a discrepancy with iperf, the results are intuitive, since Cassandra does a lot of I/O especially in CL=QUORUM. More investigation is needed here, and we will update this part of the paper as we determine the true max CPU utilization limit with Cassandra on Azure.

All tests were required to maintain non-trivial amount of storage, exceeding the available local disk on each node. We always assume for X amount of storage we need 2*(replica count)*X/#(nodes) space available on each node to allow for compaction. When the local disk was not enough, we used premium disks for both log and data.