Cosmos DB Performance

David Tucker
TECHNICAL ARCHITECT & CTO CONSULTANT
@_davidtucker_   davidtucker.net

Traditional Database Scaling

Vertical Scaling

Horizontal Scaling

Request Unit (RU)

With Cosmos DB, a Request Unit encapsulates many of the resources needed for the database into a single unit. As a baseline, one RU is equal to a 1kb item read operation from a Cosmos DB container.

Resources Encapsulated in RU’s

Processing Power (CPU)

Memory

IOPS (Input/Output Operations Per Second)

Managing Cosmos DB Throughput

Provisioned

Serverless

Provisioned Throughput

Ideal for always-on production implementation

Can be configured at the database or container

Throughput is evenly distributed to partitions

Requires 10 RU's per GB of storage

Once RU’s are consumed for a partition, future requests will be rate limited

By default, it requires a manual scaling approach to acquire more RU’s

Autoscaling (Provisioned)

With provisioned throughput, you can specify a maximum RU throughout amount, and Cosmos DB will ensure that your data is available up to that throughput amount. The minimum throughput is calculated as 10% of the maximum.

Cosmos DB Serverless

Pay only for the request units consumed and storage used

Ideal for off and on workloads such as development workloads

Currently in preview

Maximum of 5,000 RU's

Requires a new account type

Currently supports the SQL (Core) API

Best Practices

Use a partition strategy to evenly spread throughput on partitions

Provision throughput at the container for predictable performance

Use the serverless account type for development workloads

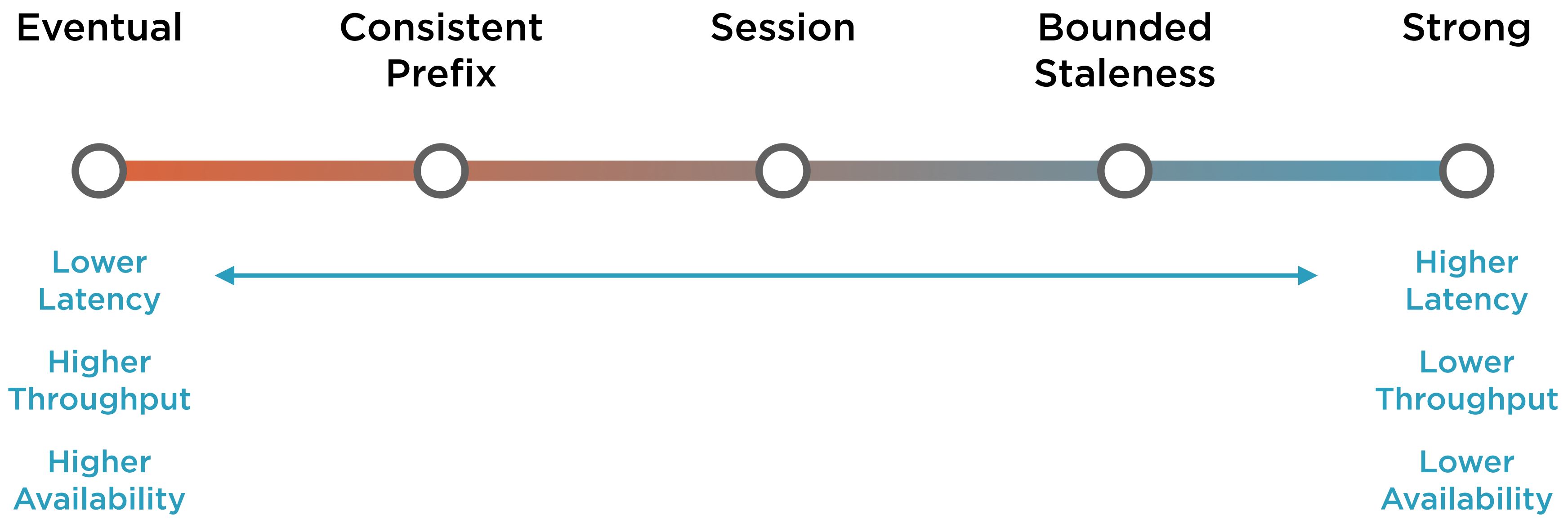
Understand the link between consistency types and the amount of RU’s consumed

Data Consistency Levels

"Distributed databases that rely on replication for high availability, low latency, or both, must make a fundamental tradeoff between the read consistency, availability, latency, and throughput."

Microsoft Cosmos DB Documentation

Consistency Level Spectrum



Consistency Levels

Strong consistency guarantees that reads get the most recent version of an item

Bounded staleness guarantees that a read has a max lag (either versions or time)

Session consistency guarantees that a client session will read its own writes

Consistent prefix consistency guarantees that updates are returned in order

Eventual consistency provides no guarantee for order

Consistency Levels for SQL API’s

Account Default

Request-specific
Level

Consistency Levels and API’s

For Gremlin and Azure Table API’s, Cosmos DB uses account default consistency level

For Cassandra writes, the account default consistency level is used

For Cassandra reads, the client consistency is mapped to a Cosmos DB level

For MongoDB, the write concern uses the account default consistency level

For MongoDB, the read concern will use a mapping to a Cosmos DB level

Throughput Considerations

Both strong and bounded staleness reads will consume twice the normal amount of request units for a request, as Cosmos DB will need to query two replicas to meet the criteria of the consistency level.

Partitioning Data in Cosmos DB

Understanding Cosmos DB Partitions

Logical Partition	**Physical Partition**
Partition Key	**Replica Set**

Logical Partition

A logical partition is a set of items within a container that share the same partition key. A container can have as many logical partitions as it needs, but each partition is limited to 20GB of storage. Logical partitions are managed by Cosmos DB, but their use is governed by your partition key strategy.

“A container is scaled by distributing data and throughput across **physical partitions**. Internally, one or more logical partitions are mapped to a single physical partition. They are entirely managed by Azure Cosmos DB.”

Microsoft Cosmos DB Documentation

Replica Set

A physical partition contains multiple replicas of the data, known as a replica set. By having this data replicated, you enable your storage to be durable and fault tolerant. These replica sets are managed by Cosmos DB.

Partition Key

Serves as the means of routing your request to the correct partition

Made up of both the key and the value of the defined partition key

Should be a value that does not change for the item

Should have many different values represented in the container

“Azure Cosmos DB uses hash-based partitioning to spread logical partitions across physical partitions. Azure Cosmos DB hashes the partition key value of an item. Then, Azure Cosmos DB allocates the key space of partition key hashes evenly across the physical partitions.”

Microsoft Cosmos DB Documentation

Strategy Considerations

Throughput is distributed evenly across all of your physical partitions

Multi-item transactions require triggers or stored procedures

You will want to minimize cross-partition queries for heavier workloads

Decide upon a partition key strategy before creating your container

Partition Key Scenario

```
{
 employeeId: 'b1af4910-b40fe0a',
 firstName: 'David',
 lastName: 'Tucker',
 email: 'david@globomantics.com',          Partition Key
 office: 'USTN1',
 department: 'development',
 reportsTo: 'b1af4910-b40fe0a'
}
```