|  |
| --- |
| **Best practices: Cosmos DB**  This document covers the best practices for Cosmos DB.  Contents  [1. Introduction 2](#_Toc19007021)  [2. Partitioning and best practices 2](#_Toc19007022)  [3. Consistency Models 8](#_Toc19007023)  [4. References 10](#_Toc19007024) |

|  |
| --- |
| Introduction |

This document covers best practices for cosmos DB

|  |
| --- |
| Partitioning and best practices |

Partitioning leveraging Azure Cosmos DB to automatically scale data across the globe. Azure Cosmos DB uses partitioning to scale individual containers in a database to meet the performance needs of your application. In partitioning, the items in a container are divided into distinct subsets called logical partitions. Logical partitions are formed based on the value of a partition key that is associated with each item in a container. All items in a logical partition have the same partition key value.

For example, a container holds items. Each item has a unique value for the UserID property. If UserID serves as the partition key for the items in the container and there are 1,000 unique UserID values, 1,000 logical partitions are created for the container.

In addition to a partition key that determines the item’s logical partition, each item in a container has an item ID (unique within a logical partition). Combining the partition key and the item ID creates the item's index, which uniquely identifies the item.

1. **Understand Partition**







What happens when partitions need to grow?

Partition Ranges can be dynamically sub-divided to seamlessly grow database as the application grows while simultaneously maintaining high availability.

Partition management is fully managed by Azure Cosmos DB, so you don't have to write code or manage your partitions.



1. **Partition Design**

It is Important to select the “Right” partition key

Partition keys acts as a **means for efficiently routing queries** and as a boundary for **multi-record** transactions.

**Key motivations are:**

* + Distribute Requests
  + Distribute Storage
  + Intelligently Route Queries for Efficiency

**Example Scenario:**

Contoso Connected Car is a vehicle telematics company. They are planning to store vehicle telemetry data from millions of vehicles every second in Azure Cosmos DB to power predictive maintenance, fleet management, and driver risk analysis.

The partition key we select will be the scope for multi-record transactions.

What are a few potential partition key choices?

* + Vehicle Model
  + Current Time
  + Device Id
  + Composite Key – Device ID + Current Time







**Partition Granularity**

Select the “Right” level of granularity for partions

Partitions should be based on most often occurring query and transactional needs. The goal is to maximize granularity and minimize cross-partition requests.



Don’t be afraid to have more partitions! More partition keys = More scalability



1. **Best Practices: Design Goals for Choosing a Good Partition Key**
   * A single logical partition has an upper limit of 10 GB of storage.
   * Distribute the overall request + storage volume
     + Avoid “hot” partition keys
   * Partition Key is scope for multi-record transactions and routing queries
     + Queries can be intelligently routed via partition key
     + Omitting partition key on query requires fan-out
   * Candidates for partition keys might include properties that appear frequently as a filter in queries. Queries can be efficiently routed by including the partition key in the filter predicate.
2. **General Tips**

* Build a POC to strengthen your understanding of the workload and iterate (avoid analyses paralysis)
* Don’t be afraid of having too many partition keys
  + Partitions keys are logical
  + More partition keys more scalability

1. **Partition Key storage limits**
   * Containers support unlimited storage by dynamically allocating additional physical partitions
   * Storage for single partition key value (logical partition) is quota'ed to 10GB.
   * When a partition key reaches its provisioned storage limit, requests to create new resources will return a HTTP Status Code of 403 (Forbidden).
   * Azure Cosmos DB will automatically add partitions, and may also return a 403 if:
     + An authorization token has expired
     + A programmatic element (UDF, Stored Procedure, Trigger) has been flagged for repeated violations

HTTP 403

1. **Design patterns for Large partition keys**
   1. **LINKED LIST APPROACH" by spreading data across incremental partition key values**

For workloads that exceed quotas for a single partition key value, you can logically spread items across multiple partition keys within a container by using a suffix on the partition key value.

As a partition fills up, you can determine when to increment the partition key value by looking for the 403 status code in your application’s logic.

HTTP 403

HTTP 403

* 1. **"CIRCULAR BUFFER" approach by resuing unique IDs**

As you insert new items into a container’s partition, you can increment the unique id for each item in the partition.

When you get a 403 status code, indicating the partition is full, you can restart your unique id and upsert the items to replace older documents.



1. **HOT/COLD Partitions**

**Partition usage can vary over time.** Partitions that are approaching thresholds are referred to as **hot**. Partitions that are underutilized are referred to as **cold**.

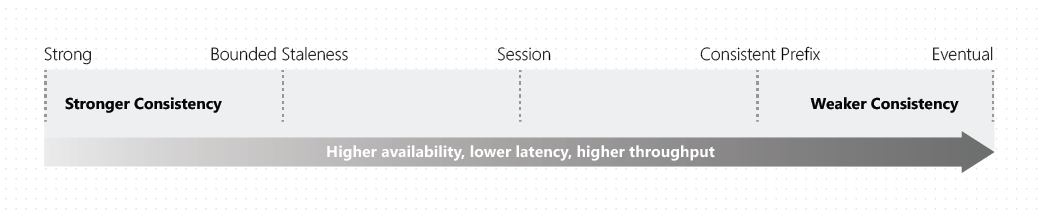


|  |
| --- |
| Consistency Models |

Distributed databases that rely on replication for high availability, low latency, or both, make the fundamental tradeoff between the read consistency vs. availability, latency, and throughput. Most commercially available distributed databases ask developers to choose between the two extreme consistency models: strong consistency and eventual consistency. The linearizability or the strong consistency model is the gold standard of data programmability. But it adds a price of higher latency (in steady state) and reduced availability (during failures). On the other hand, eventual consistency offers higher availability and better performance, but makes it hard to program applications.

Azure Cosmos DB approaches data consistency as a spectrum of choices instead of two extremes. Strong consistency and eventual consistency are at the ends of the spectrum, but there are many consistency choices along the spectrum. Developers can use these options to make precise choices and granular tradeoffs with respect to high availability and performance.

With Azure Cosmos DB, developers can choose from five well-defined consistency models on the consistency spectrum. From strongest to more relaxed, the models include strong, bounded staleness, session, consistent prefix, and eventual consistency. The models are well-defined and intuitive and can be used for specific real-world scenarios. Each model provides [availability and performance tradeoffs](https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels-tradeoffs) and is backed by the SLAs. The following image shows the different consistency levels as a spectrum.



The consistency levels are region-agnostic and are guaranteed for all operations regardless of the region from which the reads and writes are served, the number of regions associated with Azure Cosmos account, or whether account is configured with a single or multiple write regions.

The following table illustrates thes pecific guarantees each consistency level provides.

|  |  |
| --- | --- |
| **Consistency Level** | **Guarantees** |
| Strong | Linearizability (once operation is complete, it will be visible to all) |
| Bounded Staleness | Consistent Prefix.  Reads lag behind writes by at most k prefixes or t interval  Similar properties to strong consistency (except within staleness window), while preserving 99.99% availability and low latency. |
| Session | Consistent Prefix.  Within a session: monotonic reads, monotonic writes, read-your-writes, write-follows-reads  Predictable consistency for a session, high read throughput + low latency |
| Consistent Prefix | Reads will never see out of order writes (no gaps). |
| Eventual | Potential for out of order reads. Lowest cost for reads of all consistency levels. |

**Choose the right consistency level**

Distributed databases relying on replication for high availability, low latency or both, make the fundamental tradeoff between the read consistency vs. availability, latency, and throughput. Each of these consistency (strong, bounded staleness, session, consistent prefix, and eventual consistency) models is well-defined, intuitive and can be used for specific real-world scenarios. The following simple considerations will help to make the right choice in many common scenarios.

**SQL API and Table API**

Consider the following points if application is built using SQL API or Table API:

* For many real-world scenarios, session consistency is optimal and it's the recommended option. For more information, see, [How-to manage session token for your application](https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-manage-consistency#utilize-session-tokens).
* If application requires strong consistency, it is recommended to use bounded staleness consistency level.
* If requires stricter consistency guarantees than the ones provided by session consistency and single-digit-millisecond latency for writes, it is recommended to use bounded staleness consistency level.
* If application requires eventual consistency, it is recommended to use consistent prefix consistency level.
* If requires less strict consistency guarantees than the ones provided by session consistency, it is recommended to use consistent prefix consistency level.
* If requires the highest availability and the lowest latency, then use eventual consistency level.
* If requires even higher data durability without sacrificing performance, possibility to create a custom consistency level at the application layer. For more information see, [How-to implement custom synchronization in your applications](https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-custom-synchronization).

**Cassandra, MongoDB, and Gremlin APIs**

* For details on mapping between “Read Consistency Level” offered in Apache Cassandra and Cosmos DB consistency levels, see [Consistency levels and Cosmos DB APIs](https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels-across-apis#cassandra-mapping).
* For details on mapping between “Read Concern” of MongoDB and Azure Cosmos DB consistency levels, see [Consistency levels and Cosmos DB APIs](https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels-across-apis#mongo-mapping).

**Consistency guarantees in practice**

In practice, user often get stronger consistency guarantees. Consistency guarantees for a read operation correspond to the freshness and ordering of the database state that you request. Read-consistency is tied to the ordering and propagation of the write/update operations.

* When the consistency level is set to **bounded staleness**, Cosmos DB guarantees that the clients always read the value of a previous write, with a lag bounded by the staleness window.
* When the consistency level is set to **strong**, the staleness window is equivalent to zero, and the clients are guaranteed to read the latest committed value of the write operation.
* For the remaining three consistency levels, the staleness window is largely dependent on workload. For example, if there are no write operations on the database, a read operation with **eventual**, **session**, or **consistent prefix** consistency levels is likely to yield the same results as a read operation with strong consistency level.

If Azure Cosmos account is configured with a consistency level other than the strong consistency, user can find out the probability that their clients may get strong and consistent reads for their workloads by looking at the Probabilistically Bounded Staleness (PBS) metric. This metric is exposed in the Azure portal, to learn more, see [Monitor Probabilistically Bounded Staleness (PBS) metric](https://docs.microsoft.com/en-us/azure/cosmos-db/how-to-manage-consistency#monitor-probabilistically-bounded-staleness-pbs-metric).

Probabilistic bounded staleness shows how eventual is user’s eventual consistency. This metric provides an insight into how often user can get a stronger consistency than the consistency level that user have currently configured on their Azure Cosmos account. In other words, user can see the probability (measured in milliseconds) of getting strongly consistent reads for a combination of write and read regions.

|  |
| --- |
| References |

1. <https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels-choosing>
2. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels