# Partitioning and scaling in Azure DocumentDB

[Microsoft Azure DocumentDB](https://azure.microsoft.com/services/documentdb/) is designed to help you achieve fast, predictable performance and scale seamlessly along with your application as it grows. This article provides an overview of how partitioning works in DocumentDB, and describes how you can configure DocumentDB collections to effectively scale your applications.

After reading this article, you will be able to answer the following questions:

* How does partitioning work in Azure DocumentDB?
* How do I configure partitioning in DocumentDB
* What are partition keys, and how do I pick the right partition key for my application?

## Partitioning in DocumentDB

In DocumentDB, you can store and query schema-less JSON documents with order-of-millisecond response times at any scale. DocumentDB provides containers for storing data called **collections**. Collections are logical resources and can span one or more physical partitions or servers. The number of partitions is determined by DocumentDB based on the storage size and the provisioned throughput of the collection. Every partition in DocumentDB has a fixed amount of SSD-backed storage associated with it, and is replicated for high availability. Partition management is fully managed by Azure DocumentDB, and you do not have to write complex code or manage your partitions. DocumentDB collections are **practically unlimited** in terms of storage and throughput.

Partitioning is completely transparent to your application. DocumentDB supports fast reads and writes, SQL and LINQ queries, JavaScript based transactional logic, consistency levels, and fine-grained access control via REST API calls to a single collection resource. The service handles distributing data across partitions and routing query requests to the right partition.

How does this work? When you create a collection in DocumentDB, you’ll notice that there’s a **partition key property** configuration value that you can specify. This is the JSON property (or path) within your documents that can be used by DocumentDB to distribute your data among multiple servers or partitions. DocumentDB will hash the partition key value and use the hashed result to determine the partition in which the JSON document will be stored. All documents with the same partition key will be stored in the same partition.

For example, consider an application that stores data about employees and their departments in DocumentDB. Let’s choose "department" as the partition key property, in order to scale out data by department. Every document in DocumentDB must contain a mandatory "id" property that must be unique for every document with the same partition key value, e.g. "Marketing“. Every document stored in a collection must have a unique combination of partition key and id, e.g. { "Department": "Marketing", "id": "0001" }, { "Department": "Marketing", "id": "0002" }, and { "Department": "Sales", "id": "0001" }. In other words, the compound property of (partition key, id) is the primary key for your collection.

### Partition keys

The choice of the partition key is an important decision that you’ll have to make at design time. You must pick a JSON property name that has a wide range of values and is likely to have evenly distributed access patterns. The partition key is specified as a JSON path, e.g. /department represents the property department.

The following table shows examples of partition key definitions and the JSON values corresponding to each.

<tbody>  
 <tr>  
 <td valign="top"><p><strong>Partition Key Path</strong></p></td>  
 <td valign="top"><p><strong>Description</strong></p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>/department</p></td>  
 <td valign="top"><p>Corresponds to the JSON value of doc.department where doc is the document.</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>/properties/name</p></td>  
 <td valign="top"><p>Corresponds to the JSON value of doc.properties.name where doc is the document (nested property).</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>/id</p></td>  
 <td valign="top"><p>Corresponds to the JSON value of doc.id (id and partition key are the same property).</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>/"department name"</p></td>  
 <td valign="top"><p>Corresponds to the JSON value of doc["department name"] where doc is the document.</p></td>  
 </tr>  
</tbody>

[AZURE.NOTE] The syntax for partition key path is similar to the path specification for indexing policy paths with the key difference that the path corresponds to the property instead of the value, i.e. there is no wild card at the end. For example, you would specify /department/? to index the values under department, but specify /department as the partition key definition. The partition key path is implicitly indexed and cannot be excluded from indexing using indexing policy overrides.

Let’s take a look at how the choice of partition key impacts the performance of your application.

### Partitioning and provisioned throughput

DocumentDB is designed for predictable performance. When you create a collection, you reserve throughput in terms of [**request units**](documentdb-request-units.md) **(RU) per second**. Each request is assigned a request unit charge that is proportionate to the amount of system resources like CPU and IO consumed by the operation. A read of a 1 kB document with Session consistency consumes 1 request unit. A read is 1 RU regardless of the number of items stored or the number of concurrent requests running at the same. Larger documents require higher request units depending on the size. If you know the size of your entities and the number of reads you need to support for your application, you can provision the exact amount of throughput required for your application’s read needs.

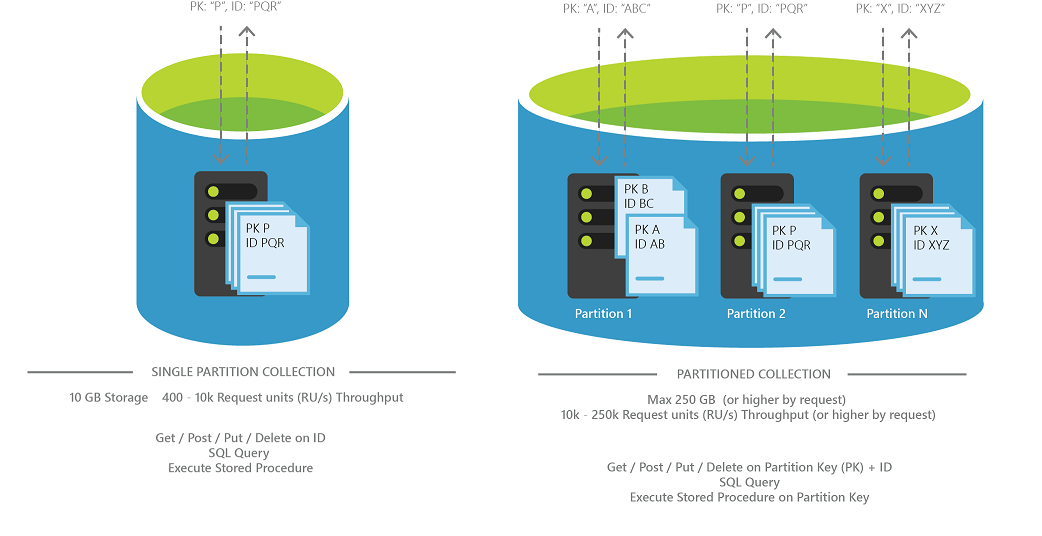
When DocumentDB stores documents, it distributes them evenly among partitions based on the partition key value. The throughput is also distributed evenly among the available partitions i.e. the throughput per partition = (total throughput per collection)/ (number of partitions).

[AZURE.TIP] In order to achieve the full throughput of the collection, you must choose a partition key that allows you to evenly distribute requests among a number of distinct partition key values.

## Single Partition and Partitioned Collections

DocumentDB supports the creation of both single-partition and partitioned collections.

* **Partitioned collections** can span multiple partitions and support very large amounts of storage and throughput. You must specify a partition key for the collection.
* **Single-partition collections** have lower price options and the ability to query and perform transactions across all collection data. They have the scalability and storage limits of a single partition. You do not have to specify a partition key for these collections.



Partitioned collections in DocumentDB

For scenarios that do not need large volumes of storage or throughput, single partition collections are a good fit. Note that single-partition collections have the scalability and storage limits of a single partition, i.e. up to 10 GB of storage and up to 10,000 request units per second.

Partitioned collections can support very large amounts of storage and throughput. The default offers however are configured to store up to 250 GB of storage and scale up to 250,000 request units per second. If you need higher storage or throughput per collection, please contact [Azure Support](/documentation/articles/documentdb-increase-limits) to have these increased for your account.

The following table lists differences in working with a single-partition and partitioned collections:

<tbody>  
 <tr>  
 <td valign="top"><p></p></td>  
 <td valign="top"><p><strong>Single Partition Collection</strong></p></td>  
 <td valign="top"><p><strong>Partitioned Collection</strong></p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>Partition Key</p></td>  
 <td valign="top"><p>None</p></td>  
 <td valign="top"><p>Required</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>Primary Key for Document</p></td>  
 <td valign="top"><p>"id"</p></td>  
 <td valign="top"><p>compound key &lt;partition key&gt; and "id"</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>Minimum Storage</p></td>  
 <td valign="top"><p>0 GB</p></td>  
 <td valign="top"><p>0 GB</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>Maximum Storage</p></td>  
 <td valign="top"><p>10 GB</p></td>  
 <td valign="top"><p>Unlimited (250 GB by default)</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>Minimum Throughput</p></td>  
 <td valign="top"><p>400 request units per second</p></td>  
 <td valign="top"><p>10,000 request units per second</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>Maximum Throughput</p></td>  
 <td valign="top"><p>10,000 request units per second</p></td>  
 <td valign="top"><p>Unlimited (250,000 request units per second by default)</p></td>  
 </tr>  
 <tr>  
 <td valign="top"><p>API versions</p></td>  
 <td valign="top"><p>All</p></td>  
 <td valign="top"><p>API 2015-12-16 and newer</p></td>  
 </tr>  
</tbody>

## Working with the SDKs

Azure DocumentDB added support for automatic partitioning with [REST API version 2015-12-16](https://msdn.microsoft.com/library/azure/dn781481.aspx). In order to create partitioned collections, you must download SDK versions 1.6.0 or newer in one of the supported SDK platforms (.NET, Node.js, Java, Python).

### Creating partitioned collections

The following sample shows a .NET snippet to create a collection to store device telemetry data of 20,000 request units per second of throughput. The SDK sets the OfferThroughput value (which in turn sets the x-ms-offer-throughput request header in the REST API). Here we set the /deviceId as the partition key. The choice of partition key is saved along with the rest of the collection metadata like name and indexing policy.

For this sample, we picked deviceId since we know that (a) since there are a large number of devices, writes can be distributed across partitions evenly and allowing us to scale the database to ingest massive volumes of data and (b) many of the requests like fetching the latest reading for a device are scoped to a single deviceId and can be retrieved from a single partition.

DocumentClient client = new DocumentClient(new Uri(endpoint), authKey);  
await client.CreateDatabaseAsync(new Database { Id = "db" });  
  
// Collection for device telemetry. Here the JSON property deviceId will be used as the partition key to   
// spread across partitions. Configured for 10K RU/s throughput and an indexing policy that supports   
// sorting against any number or string property.  
DocumentCollection myCollection = new DocumentCollection();  
myCollection.Id = "coll";  
myCollection.PartitionKey.Paths.Add("/deviceId");  
  
await client.CreateDocumentCollectionAsync(  
 UriFactory.CreateDatabaseUri("db"),  
 myCollection,  
 new RequestOptions { OfferThroughput = 20000 });

[AZURE.NOTE] In order to create partitioned collections, you must specify a throughput value of > 10,000 request units per second. Since throughput is in multiples of 100, this has to be 10,100 or higher.

This method makes a REST API call to DocumentDB, and the service will provision a number of partitions based on the requested throughput. You can change the throughput of a collection as your performance needs evolve. See [Performance Levels](/documentation/articles/documentdb-performance-levels) for more details.

### Reading and writing documents

Now, let’s insert data into DocumentDB. Here’s a sample class containing a device reading, and a call to CreateDocumentAsync to insert a new device reading into a collection.

public class DeviceReading  
{  
 [JsonProperty("id")]  
 public string Id;  
  
 [JsonProperty("deviceId")]  
 public string DeviceId;  
  
 [JsonConverter(typeof(IsoDateTimeConverter))]  
 [JsonProperty("readingTime")]  
 public DateTime ReadingTime;  
  
 [JsonProperty("metricType")]  
 public string MetricType;  
  
 [JsonProperty("unit")]  
 public string Unit;  
  
 [JsonProperty("metricValue")]  
 public double MetricValue;  
 }  
  
// Create a document. Here the partition key is extracted as "XMS-0001" based on the collection definition  
await client.CreateDocumentAsync(  
 UriFactory.CreateDocumentCollectionUri("db", "coll"),  
 new DeviceReading  
 {  
 Id = "XMS-001-FE24C",  
 DeviceId = "XMS-0001",  
 MetricType = "Temperature",  
 MetricValue = 105.00,  
 Unit = "Fahrenheit",  
 ReadingTime = DateTime.UtcNow  
 });

Let’s read the document by it’s partition key and id, update it, and then as a final step, delete it by partition key and id. Note that the reads include a PartitionKey value (corresponding to the x-ms-documentdb-partitionkey request header in the REST API).

// Read document. Needs the partition key and the ID to be specified  
Document result = await client.ReadDocumentAsync(  
 UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),   
 new RequestOptions { PartitionKey = new PartitionKey("XMS-0001") });  
  
DeviceReading reading = (DeviceReading)(dynamic)result;  
  
// Update the document. Partition key is not required, again extracted from the document  
reading.MetricValue = 104;  
reading.ReadingTime = DateTime.UtcNow;  
  
await client.ReplaceDocumentAsync(  
 UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),   
 reading);  
  
// Delete document. Needs partition key  
await client.DeleteDocumentAsync(  
 UriFactory.CreateDocumentUri("db", "coll", "XMS-001-FE24C"),   
 new RequestOptions { PartitionKey = new PartitionKey("XMS-0001") });

### Querying partitioned collections

When you query data in partitioned collections, DocumentDB automatically routes the query to the partitions corresponding to the partition key values specified in the filter (if there are any). For example, this query is routed to just the partition containing the partition key “XMS-0001”.

// Query using partition key  
IQueryable<DeviceReading> query = client.CreateDocumentQuery<DeviceReading>(  
 UriFactory.CreateDocumentCollectionUri("db", "coll"))  
 .Where(m => m.MetricType == "Temperature" && m.DeviceId == "XMS-0001");

The following query does not have a filter on the partition key (DeviceId) and is fanned out to all partitions where it is executed against the partition’s index. Note that you have to specify the EnableCrossPartitionQuery (x-ms-documentdb-query-enablecrosspartition in the REST API) to have the SDK to execute a query across partitions.

// Query across partition keys  
IQueryable<DeviceReading> crossPartitionQuery = client.CreateDocumentQuery<DeviceReading>(  
 UriFactory.CreateDocumentCollectionUri("db", "coll"),   
 new FeedOptions { EnableCrossPartitionQuery = true })  
 .Where(m => m.MetricType == "Temperature" && m.MetricValue > 100);

### Executing stored procedures

You can also execute atomic transactions against documents with the same device ID, e.g. if you’re maintaining aggregates or the latest state of a device in a single document.

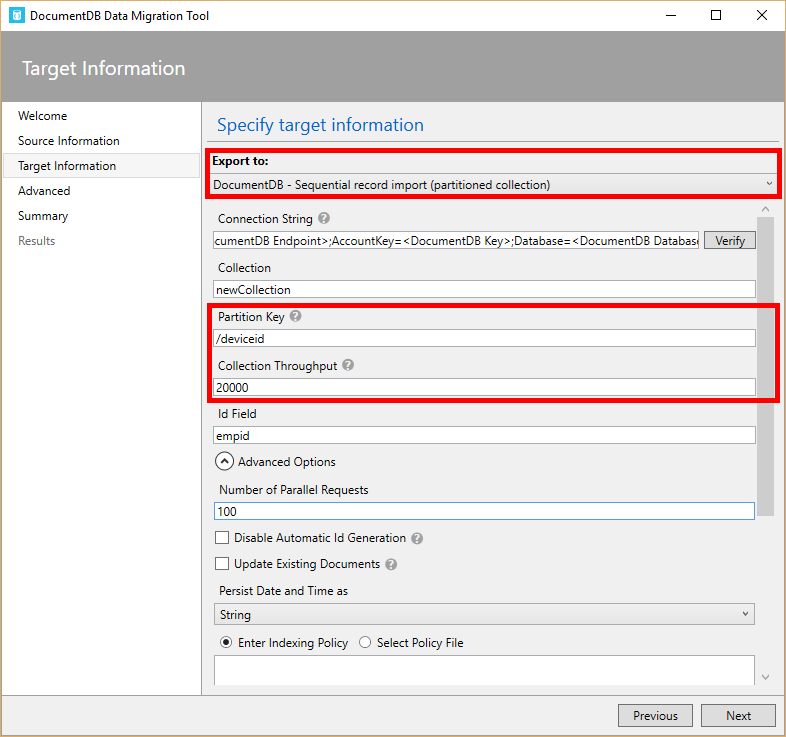
await client.ExecuteStoredProcedureAsync<DeviceReading>(  
 UriFactory.CreateStoredProcedureUri("db", "coll", "SetLatestStateAcrossReadings"),  
 "XMS-001-FE24C",  
 new RequestOptions { PartitionKey = new PartitionKey("XMS-001") });

In the next section, we look at how you can move to partitioned collections from single-partition collections.

### Migrating from single-partition to partitioned collections When an application using a single-partition collection needs higher throughput (>10,000 RU/s) or larger data storage (>10GB), you can use the [DocumentDB Data Migration Tool](http://www.microsoft.com/downloads/details.aspx?FamilyID=cda7703a-2774-4c07-adcc-ad02ddc1a44d) to migrate the data from the single-partition collection to a partitioned collection.

To migrate from a single-partition collection to a partitioned collection

1. Export data from the single-partition collection to JSON. See [Export to JSON file](/documentation/articles/documentdb-import-data#export-to-json-file) for additional details.
2. Import the data into a partitioned collection created with a partition key definition and over 10,000 request units per second throughput, as shown in the example below. See [Import to DocumentDB](/documentation/articles/documentdb-import-data#DocumentDBSeqTarget) for additional details.



Migrating Data to a Partitioned collection in DocumentDB

[AZURE.TIP] For faster import times, consider increasing the Number of Parallel Requests to 100 or higher to take advantage of the higher throughput available for partitioned collections.

Now that we’ve completed the basics, let’s look at a few important design considerations when working with partition keys in DocumentDB.

## Designing for partitioning

The choice of the partition key is an important decision that you’ll have to make at design time. This section describes some of the tradeoffs involved in selecting a partition key for your collection.

### Partition key as the transaction boundary

Your choice of partition key should balance the need to enable the use of transactions against the requirement to distribute your entities across multiple partition keys to ensure a scalable solution. At one extreme, you could set the same partition key for all your documents, but this may limit the scalability of your solution. At the other extreme, you could assign a unique partition key for each document, which would be highly scalable but would prevent you from using cross document transactions via stored procedures and triggers. An ideal partition key is one that enables you to use efficient queries and that has sufficient cardinality to ensure your solution is scalable.

### Avoiding storage and performance bottlenecks

It is also important to pick a property which allows writes to be distributed across a number of distinct values. Requests to the same partition key cannot exceed the throughput of a single partition, and will be throttled. So it is important to pick a partition key that does not result in **“hot spots”** within your application. The total storage size for documents with the same partition key can also not exceed 10 GB in storage.

### Examples of good partition keys

Here are a few examples for how to pick the partition key for your application:

* If you’re implementing a user profile backend, then the user ID is a good choice for partition key.
* If you’re storing IoT data e.g. device state, a device ID is a good choice for partition key.
* If you’re using DocumentDB for logging time-series data, then the hostname or process ID is a good choice for partition key.
* If you have a multi-tenant architecture, the tenant ID is a good choice for partition key.

Note that in some use cases (like the IoT and user profiles described above), the partition key might be the same as your id (document key). In others like the time series data, you might have a partition key that’s different than the id.

### Partitioning and logging/time-series data

One of the most common use cases of DocumentDB is for logging and telemetry. It is important to pick a good partition key since you might need to read/write vast volumes of data. The choice will depend on your read and write rates and kinds of queries you expect to run. Here are some tips on how to choose a good partition key.

* If your use case involves a small rate of writes acculumating over a long period of time, and need to query by ranges of timestamps and other filters, then using a rollup of the timestamp e.g. date as a partition key is a good approach. This allows you to query over all the data for a date from a single partition.
* If your workload is write heavy, which is generally more common, you should use a partition key that’s not based on timestamp so that DocumentDB can distribute writes evenly across a number of partitions. Here a hostname, process ID, activity ID, or another property with high cardinality is a good choice.
* A third approach is a hybrid one where you have multiple collections, one for each day/month and the partition key is a granular property like hostname. This has the benefit that you can set different performance levels based on the time window, e.g. the collection for the current month is provisioned with higher throughput since it serves reads and writes, whereas previous months with lower throughput since they only serve reads.

### Partitioning and multi-tenancy

If you are implementing a multi-tenant application using DocumentDB, there are two major patterns for implementing tenancy with DocumentDB – one partition key per tenant, and one collection per tenant. Here are the pros and cons for each:

* One Partition Key per tenant: In this model, tenants are collocated within a single collection. But queries and inserts for documents within a single tenant can be performed against a single partition. You can also implement transactional logic across all documents within a tenant. Since multiple tenants share a collection, you can save storage and throughput costs by pooling resources for tenants within a single collection rather than provisioning extra headroom for each tenant. The drawback is that you do not have performance isolation per tenant. Performance/throughput increases apply to the entire collection vs targeted increases for tenants.
* One Collection per tenant: Each tenant has its own collection. In this model, you can reserve performance per tenant. With DocumentDB’s new consumption based pricing model, this model is more cost-effective for multi-tenant applications with a small number of tenants.

You can also use a combination/tiered approach that collocates small tenants and migrates larger tenants to their own collection.

## Next Steps

In this article, we’ve described how partitioning works in Azure DocumentDB, how you can create partitioned collections, and how you can pick a good partition key for your application.

* Get started coding with the [SDKs](/documentation/articles/documentdb-sdk-dotnet) or the [REST API](https://msdn.microsoft.com/library/azure/dn781481.aspx)
* Learn about [provisioned throughput in DocumentDB](/documentation/articles/documentdb-performance-levels)
* If you would like to customize how your application performs partitioning, you can plug in your own client-side partitioning implementation. See [Client-side partitioning support](/documentation/articles/documentdb-sharding).