# DocumentDB indexing policies

While many customers are happy to let DocumentDB automatically handle [all aspects of indexing](documentdb-indexing.md), DocumentDB also supports specifying a custom **indexing policy** for collections during creation. Indexing policies in DocumentDB are more flexible and powerful than secondary indexes offered in other database platforms, because they let you design and customize the shape of the index without sacrificing schema flexibility. To learn how indexing works within DocumentDB, you must understand that by managing indexing policy, you can make fine-grained tradeoffs between index storage overhead, write and query throughput, and query consistency.

In this article, we take a close look at DocumentDB indexing policies, how you can customize indexing policy, and the associated trade-offs.

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

* How can I override the properties to include or exclude from indexing?
* How can I configure the index for eventual updates?
* How can I configure indexing to perform Order By or range queries?
* How do I make changes to a collection’s indexing policy?
* How do I compare storage and performance of different indexing policies?

## Customizing the indexing policy of a collection

Developers can customize the trade-offs between storage, write/query performance, and query consistency, by overriding the default indexing policy on a DocumentDB collection and configuring the following aspects.

* **Including/Excluding documents and paths to/from index**. Developers can choose certain documents to be excluded or included in the index at the time of inserting or replacing them to the collection. Developers can also choose to include or exclude certain JSON properties a.k.a. paths (including wildcard patterns) to be indexed across documents which are included in an index.
* **Configuring Various Index Types**. For each of the included paths, developers can also specify the type of index they require over a collection based on their data and expected query workload and the numeric/string “precision” for each path.
* **Configuring Index Update Modes**. DocumentDB supports three indexing modes which can be configured via the indexing policy on a DocumentDB collection: Consistent, Lazy and None.

The following .NET code snippet shows how to set a custom indexing policy during the creation of a collection. Here we set the policy with Range index for strings and numbers at the maximum precision. This policy lets us execute Order By queries against strings.

DocumentCollection collection = new DocumentCollection { Id = "myCollection" };  
  
collection.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.String) { Precision = -1 });  
collection.IndexingPolicy.IndexingMode = IndexingMode.Consistent;  
  
await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), collection);

[AZURE.NOTE] The JSON schema for indexing policy was changed with the release of REST API version 2015-06-03 to support Range indexes against strings. .NET SDK 1.2.0 and Java, Python, and Node.js SDKs 1.1.0 support the new policy schema. Older SDKs use the REST API version 2015-04-08 and support the older schema of Indexing Policy.

By default, DocumentDB indexes all string properties within documents consistently with a Hash index, and numeric properties with a Range index.

### Database indexing modes

DocumentDB supports three indexing modes which can be configured via the indexing policy on a DocumentDB collection – Consistent, Lazy and None.

**Consistent**: If a DocumentDB collection’s policy is designated as “consistent”, the queries on a given DocumentDB collection follow the same consistency level as specified for the point-reads (i.e. strong, bounded-staleness, session or eventual). The index is updated synchronously as part of the document update (i.e. insert, replace, update, and delete of a document in a DocumentDB collection). Consistent indexing supports consistent queries at the cost of possible reduction in write throughput. This reduction is a function of the unique paths that need to be indexed and the “consistency level”. Consistent indexing mode is designed for “write quickly, query immediately” workloads.

**Lazy**: To allow maximum document ingestion throughput, a DocumentDB collection can be configured with lazy consistency; meaning queries are eventually consistent. The index is updated asynchronously when a DocumentDB collection is quiescent i.e. when the collection’s throughput capacity is not fully utilized to serve user requests. For “ingest now, query later” workloads requiring unhindered document ingestion, “lazy” indexing mode may be suitable.

**None**: A collection marked with index mode of “None” has no index associated with it. This is commonly used if DocumentDB is utilized as a key-value storage and documents are accessed only by their ID property.

[AZURE.NOTE] Configuring the indexing policy with “None” has the side effect of dropping any existing index. Use this if your access patterns are only require “id” and/or “self-link”.

The following sample show how create a DocumentDB collection using the .NET SDK with consistent automatic indexing on all document insertions.

The following table shows the consistency for queries based on the indexing mode (Consistent and Lazy) configured for the collection and the consistency level specified for the query request. This applies to queries made using any interface - REST API, SDKs or from within stored procedures and triggers.

<tbody>  
 <tr>  
 <td valign="top">  
 <p>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 <strong>Consistent</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 <strong>Lazy</strong>  
 </p>  
 </td>   
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Strong</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Strong  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 </tr>   
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Bounded Staleness</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Bounded Staleness  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 </tr>   
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Session</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Session  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 </tr>   
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Eventual</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 </tr>   
</tbody>

DocumentDB returns an error for queries made on collections with None indexing mode. Queries can still be executed as scans via the explicit x-ms-documentdb-enable-scans header in the REST API or the EnableScanInQuery request option using the .NET SDK. Some query features like ORDER BY are not supported as scans with EnableScanInQuery.

The following table shows the consistency for queries based on the indexing mode (Consistent, Lazy, and None) when EnableScanInQuery is specified.

<tbody>  
 <tr>  
 <td valign="top">  
 <p>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 <strong>Consistent</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 <strong>Lazy</strong>  
 </p>  
 </td>   
 <td valign="top">  
 <p>  
 <strong>None</strong>  
 </p>  
 </td>   
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Strong</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Strong  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 <td valign="top">  
 <p>  
 Strong  
 </p>  
 </td>   
 </tr>   
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Bounded Staleness</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Bounded Staleness  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 <td valign="top">  
 <p>  
 Bounded Staleness  
 </p>  
 </td>   
 </tr>   
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Session</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Session  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 <td valign="top">  
 <p>  
 Session  
 </p>  
 </td>   
 </tr>   
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Eventual</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 <td valign="top">  
 <p>  
 Eventual  
 </p>  
 </td>   
 </tr>   
</tbody>

The following code sample show how create a DocumentDB collection using the .NET SDK with consistent indexing on all document insertions.

// Default collection creates a hash index for all string and numeric   
 // fields. Hash indexes are compact and offer efficient  
 // performance for equality queries.  
   
 var collection = new DocumentCollection { Id ="defaultCollection" };  
   
 collection.IndexingPolicy.IndexingMode = IndexingMode.Consistent;  
   
 collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("mydb"), collection);

### Index paths

DocumentDB models JSON documents and the index as trees, and allows you to tune to policies for paths within the tree. You can find more details in this [introduction to DocumentDB indexing](/documentation/articles/documentdb-indexing). Within documents, you can choose which paths must be included or excluded from indexing. This can offer improved write performance and lower index storage for scenarios when the query patterns are known beforehand.

Index paths start with the root (/) and typically end with the ? wildcard operator, denoting that there are multiple possible values for the prefix. For example, to serve SELECT \* FROM Families F WHERE F.familyName = “Andersen”, you must include an index path for /familyName/? in the collection’s index policy.

Index paths can also use the \* wildcard operator to specify the behavior for paths recursively under the prefix. For example, /payload/\* can be used to exclude everything under the payload property from indexing.

Here are the common patterns for specifying index paths:

<tbody>  
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Path</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 <strong>Description/use case</strong>  
 </p>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 /\*  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Default path for collection. Recursive and applies to whole document tree.  
 </p>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 /prop/?  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Index path required to serve queries like the following (with Hash or Range types respectively):  
 </p>  
 <p>  
 SELECT \* FROM collection c WHERE c.prop = "value"  
 </p>  
 <p>  
 SELECT \* FROM collection c WHERE c.prop &gt; 5  
 </p>  
 <p>  
 SELECT \* FROM collection c ORDER BY c.prop  
 </p>   
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 /prop/\*  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Index path for all paths under the specified label. Works with the following queries  
 </p>  
 <p>  
 SELECT \* FROM collection c WHERE c.prop = "value"  
 </p>  
 <p>  
 SELECT \* FROM collection c WHERE c.prop.subprop &gt; 5  
 </p>  
 <p>  
 SELECT \* FROM collection c WHERE c.prop.subprop.nextprop = "value"  
 </p>  
 <p>  
 SELECT \* FROM collection c ORDER BY c.prop  
 </p>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 /props/[]/?  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Index path required to serve iteration and JOIN queries against arrays of scalars like ["a", "b", "c"]:  
 </p>  
 <p>  
 SELECT tag FROM tag IN collection.props WHERE tag = "value"  
 </p>  
 <p>  
 SELECT tag FROM collection c JOIN tag IN c.props WHERE tag > 5  
 </p>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 /props/[]/subprop/?  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Index path required to serve iteration and JOIN queries against arrays of objects like [{subprop: "a"}, {subprop: "b"}]:  
 </p>  
 <p>  
 SELECT tag FROM tag IN collection.props WHERE tag.subprop = "value"  
 </p>  
 <p>  
 SELECT tag FROM collection c JOIN tag IN c.props WHERE tag.subprop = "value"  
 </p>  
 </td>  
 </tr>   
 <tr>  
 <td valign="top">  
 <p>  
 /prop/subprop/?  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Index path required to serve queries (with Hash or Range types respectively):  
 </p>  
 <p>  
 SELECT \* FROM collection c WHERE c.prop.subprop = "value"  
 </p>  
 <p>  
 SELECT \* FROM collection c WHERE c.prop.subprop &gt; 5  
 </p>  
 <p>  
 SELECT \* FROM collection c ORDER BY c.prop.subprop  
 </p>   
 </td>  
 </tr>  
</tbody>

[AZURE.NOTE] While setting custom index paths, you are required to specify the default indexing rule for the entire document tree denoted by the special path “/\*“.

The following example configures a specific path with range indexing and a custom precision value of 20 bytes:

var collection = new DocumentCollection { Id = "rangeSinglePathCollection" };   
  
collection.IndexingPolicy.IncludedPaths.Add(  
 new IncludedPath {   
 Path = "/Title/?",   
 Indexes = new Collection<Index> {   
 new RangeIndex(DataType.String) { Precision = 20 } }   
 });  
  
// Default for everything else  
collection.IndexingPolicy.IncludedPaths.Add(  
 new IncludedPath {   
 Path = "/\*" ,  
 Indexes = new Collection<Index> {  
 new HashIndex(DataType.String) { Precision = 3 },   
 new RangeIndex(DataType.Number) { Precision = -1 }   
 }  
 });  
   
collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), pathRange);

### Index data types, kinds and precisions

Now that we’ve taken a look at how to specify paths, let’s look at the options we can use to configure the indexing policy for a path. You can specify one or more indexing definitions for every path:

* Data type: **String**, **Number** or **Point** (can contain only one entry per data type per path)
* Index kind: **Hash** (equality queries), **Range** (equality, range or Order By queries), or **Spatial** (spatial queries)
* Precision: 1-8 or -1 (Maximum precision) for numbers, 1-100 (Maximum precision) for string

#### Index kind

DocumentDB supports Hash and Range index kinds for every path (that can configured for strings, numbers or both).

* **Hash** supports efficient equality and JOIN queries. For most use cases, hash indexes do not need a higher precision than the default value of 3 bytes.
* **Range** supports efficient equality queries, range queries (using >, <, >=, <=, !=), and Order By queries. Order By queries by default also require maximum index precision (-1).

DocumentDB also supports the Spatial index kind for every path, that can be specified for the Point data type. The value at the specified path must be a valid GeoJSON point like {"type": "Point", "coordinates": [0.0, 10.0]}.

* **Spatial** supports efficient spatial (within and distance) queries.

Here are the supported index kinds and examples of queries that they can be used to serve:

<tbody>  
 <tr>  
 <td valign="top">  
 <p>  
 <strong>Index kind</strong>  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 <strong>Description/use case</strong>  
 </p>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 Hash  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Hash over /prop/? (or /\*) can be used to serve the following queries efficiently:  
 SELECT \* FROM collection c WHERE c.prop = "value"  
 Hash over /props/[]/? (or /\* or /props/\*) can be used to serve the following queries efficiently:  
 SELECT tag FROM collection c JOIN tag IN c.props WHERE tag = 5  
 </p>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 Range  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Range over /prop/? (or /\*) can be used to serve the following queries efficiently:  
 SELECT \* FROM collection c WHERE c.prop = "value"  
 SELECT \* FROM collection c WHERE c.prop > 5  
 SELECT \* FROM collection c ORDER BY c.prop  
 </p>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <p>  
 Spatial  
 </p>  
 </td>  
 <td valign="top">  
 <p>  
 Range over /prop/? (or /\*) can be used to serve the following queries efficiently:  
 SELECT \* FROM collection c   
 WHERE ST\_DISTANCE(c.prop, {"type": "Point", "coordinates": [0.0, 10.0]}) < 40  
 SELECT \* FROM collection c WHERE ST\_WITHIN(c.prop, {"type": "Polygon", ... })  
 </p>  
 </td>  
 </tr>   
</tbody>

By default, an error is returned for queries with range operators such as >= if there is no range index (of any precision) in order to signal that a scan might be necessary to serve the query. Range queries can be performed without a range index using the x-ms-documentdb-enable-scans header in the REST API or the EnableScanInQuery request option using the .NET SDK. If there are any other filters in the query that DocumentDB can use the index to filter against, then no error will be returned.

The same rules apply for spatial queries. By default, an error is returned for spatial queries if there is no spatial index, and there are no other filters that can be served from the index. They can be performed as a scan using x-ms-documentdb-enable-scan/EnableScanInQuery.

#### Index precision

Index precision lets you tradeoff between index storage overhead and query performance. For numbers, we recommend using the default precision configuration of -1 (“maximum”). Since numbers are 8 bytes in JSON, this is equivalent to a configuration of 8 bytes. Picking a lower value for precision, such as 1-7, means that values within some ranges map to the same index entry. Therefore you will reduce index storage space, but query execution might have to process more documents and consequently consume more throughput i.e., request units.

Index precision configuration has more practical application with string ranges. Since strings can be any arbitrary length, the choice of the index precision can impact the performance of string range queries, and impact the amount of index storage space required. String range indexes can be configured with 1-100 or -1 (“maximum”). If you would like to perform Order By queries against string properties, then you must specify a precision of -1 for the corresponding paths.

Spatial indexes always use the default index precision for points and cannot be overriden.

The following example shows how to increase the precision for range indexes in a collection using the .NET SDK.

**Create a collection with a custom index precision**

var rangeDefault = new DocumentCollection { Id = "rangeCollection" };  
  
// Override the default policy for Strings to range indexing and "max" (-1) precision  
rangeDefault.IndexingPolicy = new IndexingPolicy(new RangeIndex(DataType.String) { Precision = -1 });  
  
await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), rangeDefault);

[AZURE.NOTE] DocumentDB returns an error when a query uses Order By but does not have a range index against the queried path with the maximum precision.

Similarly, paths can be completely excluded from indexing. The next example shows how to exclude an entire section of the documents (a.k.a. a sub-tree) from indexing using the “\*" wildcard.

var collection = new DocumentCollection { Id = "excludedPathCollection" };  
collection.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/\*" });  
collection.IndexingPolicy.ExcludedPaths.Add(new ExcludedPath { Path = "/nonIndexedContent/\*");  
  
collection = await client.CreateDocumentCollectionAsync(UriFactory.CreateDatabaseUri("db"), excluded);

## Opting in and opting out of indexing

You can choose whether you want the collection to automatically index all documents. By default, all documents are automatically indexed, but you can choose to turn it off. When indexing is turned off, documents can be accessed only through their self-links or by queries using ID.

With automatic indexing turned off, you can still selectively add only specific documents to the index. Conversely, you can leave automatic indexing on and selectively choose to exclude only specific documents. Indexing on/off configurations are useful when you have only a subset of documents that need to be queried.

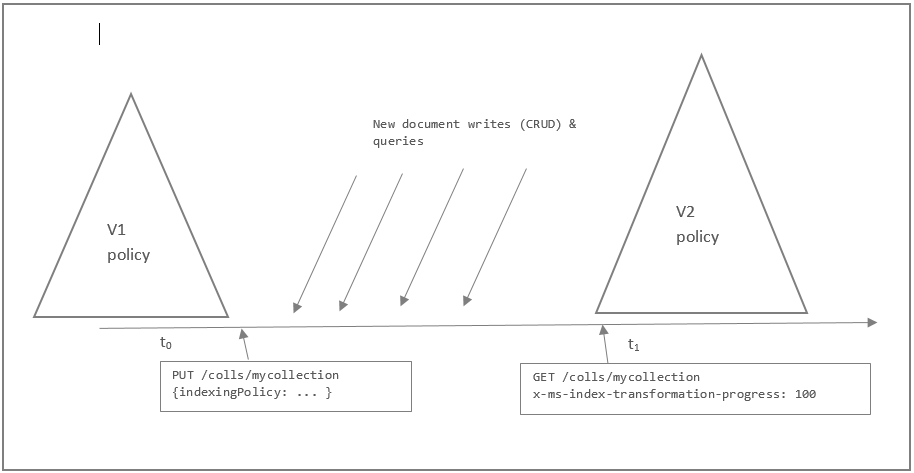
For example, the following sample shows how to include a document explicitly using the [DocumentDB .NET SDK](https://github.com/Azure/azure-documentdb-java) and the [RequestOptions.IndexingDirective](http://msdn.microsoft.com/library/microsoft.azure.documents.client.requestoptions.indexingdirective.aspx) property.

// If you want to override the default collection behavior to either  
// exclude (or include) a Document from indexing,  
// use the RequestOptions.IndexingDirective property.  
client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"),  
 new { id = "AndersenFamily", isRegistered = true },  
 new RequestOptions { IndexingDirective = IndexingDirective.Include });

## Modifying the indexing policy of a collection

DocumentDB allows you to make changes to the indexing policy of a collection on the fly. A change in indexing policy on a DocumentDB collection can lead to a change in the shape of the index including the paths can be indexed, their precision, as well as the consistency model of the index itself. Thus a change in indexing policy, effectively requires a transformation of the old index into a new one.

**Online Index Transformations**



How indexing works – DocumentDB online index transformations

Index transformations are made online, meaning that the documents indexed per the old policy are efficiently transformed per the new policy **without affecting the write availability or the provisioned throughput** of the collection. The consistency of read and write operations made using the REST API, SDKs or from within stored procedures and triggers is not impacted during index transformation. This means that there is no performance degradation or downtime to your apps when you make an indexing policy change.

However, during the time that index transformation is progress, queries are eventually consistent regardless of the indexing mode configuration (Consistent or Lazy). This also applies to queries from all interfaces – REST API, SDKs, and from within stored procedures and triggers. Just like with Lazy indexing, index transformation is performed asynchronously in the background on the replicas using the spare resources available for a given replica.

Index transformations are also made **in-situ** (in place), i.e. DocumentDB does not maintain two copies of the index and swap the old index out with the new one. This means that no additional disk space is required or consumed in your collections while performing index transformations.

When you change indexing policy, how the changes are applied to move from the old index to the new one depend primarily on the indexing mode configurations more so than the other values like included/excluded paths, index kinds and precisions. If both your old and new policies use consistent indexing, then DocumentDB performs an online index transformation. You cannot apply another indexing policy change with consistent indexing mode while the transformation is in progress.

You can however move to Lazy or None indexing mode while a transformation is in progress.

* When you move to Lazy, the index policy change is made effective immediately and DocumentDB starts recreating the index asynchronously.
* When you move to None, then the index is dropped effective immediately. Moving to None is useful when you want to cancel an in progress transformation and start fresh with a different indexing policy.

If you’re using the .NET SDK, you can kick of an indexing policy change using the new **ReplaceDocumentCollectionAsync** method and track the percentage progress of the index transformation using the **IndexTransformationProgress** response property from a **ReadDocumentCollectionAsync** call. Other SDKs and the REST API support equivalent properties and methods for making indexing policy changes.

Here’s a code snippet that shows how to modify a collection’s indexing policy from Consistent indexing mode to Lazy.

**Modify Indexing Policy from Consistent to Lazy**

// Switch to lazy indexing.  
Console.WriteLine("Changing from Default to Lazy IndexingMode.");  
  
collection.IndexingPolicy.IndexingMode = IndexingMode.Lazy;  
  
await client.ReplaceDocumentCollectionAsync(collection);

You can check the progress of an index transformation by calling ReadDocumentCollectionAsync, for example, as shown below.

**Track Progress of Index Transformation**

long smallWaitTimeMilliseconds = 1000;  
long progress = 0;  
  
while (progress < 100)  
{  
 ResourceResponse<DocumentCollection> collectionReadResponse = await client.ReadDocumentCollectionAsync(  
 UriFactory.CreateDocumentCollectionUri("db", "coll"));  
  
 progress = collectionReadResponse.IndexTransformationProgress;  
  
 await Task.Delay(TimeSpan.FromMilliseconds(smallWaitTimeMilliseconds));  
}

You can drop the index for a collection by moving to the None indexing mode. This might be a useful operational tool if you want to cancel an in-progress transformation and start a new one immediately.

**Dropping the index for a collection**

// Switch to lazy indexing.  
Console.WriteLine("Dropping index by changing to to the None IndexingMode.");  
  
collection.IndexingPolicy.IndexingMode = IndexingMode.None;  
  
await client.ReplaceDocumentCollectionAsync(collection);

When would you make indexing policy changes to your DocumentDB collections? The following are the most common use cases:

* Serve consistent results during normal operation, but fall back to lazy indexing during bulk data imports
* Start using new indexing features on your current DocumentDB collections, e.g., like geospatial querying which require the Spatial index kind, or Order By/string range queries which require the string Range index kind
* Hand select the properties to be indexed and change them over time
* Tune indexing precision to improve query performance or reduce storage consumed

[AZURE.NOTE] To modify indexing policy using ReplaceDocumentCollectionAsync, you need version >= 1.3.0 of the .NET SDK

For index transformation to complete successfully, you must ensure that there is sufficient free storage space available on the collection. If the collection reaches its storage quota, then the index transformation will be paused. Index transformation will automatically resume once storage space is available, e.g. if you delete some documents.

## Performance tuning

The DocumentDB APIs provide information about performance metrics such as the index storage used, and the throughput cost (request units) for every operation. This information can be used to compare various indexing policies and for performance tuning.

To check the storage quota and usage of a collection, run a HEAD or GET request against the collection resource, and inspect the x-ms-request-quota and the x-ms-request-usage headers. In the .NET SDK, the [DocumentSizeQuota](http://msdn.microsoft.com/library/dn850325.aspx) and [DocumentSizeUsage](http://msdn.microsoft.com/library/azure/dn850324.aspx) properties in [ResourceResponse](http://msdn.microsoft.com/library/dn799209.aspx) contain these corresponding values.

// Measure the document size usage (which includes the index size) against   
 // different policies.  
 ResourceResponse<DocumentCollection> collectionInfo = await client.ReadDocumentCollectionAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"));   
 Console.WriteLine("Document size quota: {0}, usage: {1}", collectionInfo.DocumentQuota, collectionInfo.DocumentUsage);

To measure the overhead of indexing on each write operation (create, update, or delete), inspect the x-ms-request-charge header (or the equivalent [RequestCharge](http://msdn.microsoft.com/library/dn799099.aspx) property in [ResourceResponse](http://msdn.microsoft.com/library/dn799209.aspx) in the .NET SDK) to measure the number of request units consumed by these operations.

// Measure the performance (request units) of writes.   
 ResourceResponse<Document> response = await client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri("db", "coll"), myDocument);   
 Console.WriteLine("Insert of document consumed {0} request units", response.RequestCharge);  
   
 // Measure the performance (request units) of queries.   
 IDocumentQuery<dynamic> queryable = client.CreateDocumentQuery(UriFactory.CreateDocumentCollectionUri("db", "coll"), queryString).AsDocumentQuery();  
  
 double totalRequestCharge = 0;  
 while (queryable.HasMoreResults)  
 {  
 FeedResponse<dynamic> queryResponse = await queryable.ExecuteNextAsync<dynamic>();   
 Console.WriteLine("Query batch consumed {0} request units",queryResponse.RequestCharge);  
 totalRequestCharge += queryResponse.RequestCharge;  
 }  
   
 Console.WriteLine("Query consumed {0} request units in total", totalRequestCharge);

## Changes to the indexing policy specification

A change in the schema for indexing policy was introduced on July 7, 2015 with REST API version 2015-06-03. The corresponding classes in the SDK versions have new implementations to match the schema.

The following changes were implemented in the JSON specification:

* Indexing Policy supports Range indexes for strings
* Each path can have multiple index definitions, one for each data type
* Indexing precision supports 1-8 for numbers, 1-100 for strings, and -1 (maximum precision)
* Paths segments do not require a double quotation to escape each path. For example, you can add a path for /title/? instead of /“title”/?
* The root path representing “all paths” can be represented as /\* (in addition to /)

If you have code that provisions collections with a custom indexing policy written with version 1.1.0 of the .NET SDK or older, you will need to change your application code to handle these changes in order to move to SDK version 1.2.0. If you do not have code that configures indexing policy, or plan to continue using an older SDK version, no changes are required.

For a practical comparison, here is one example custom indexing policy written using the REST API version 2015-06-03 as well as the previous version 2015-04-08.

**Previous Indexing Policy JSON**

{  
 "automatic":true,  
 "indexingMode":"Consistent",  
 "IncludedPaths":[  
 {  
 "IndexType":"Hash",  
 "Path":"/",  
 "NumericPrecision":7,  
 "StringPrecision":3  
 }  
 ],  
 "ExcludedPaths":[  
 "/\"nonIndexedContent\"/\*"  
 ]  
}

**Current Indexing Policy JSON**

{  
 "automatic":true,  
 "indexingMode":"Consistent",  
 "includedPaths":[  
 {  
 "path":"/\*",  
 "indexes":[  
 {  
 "kind":"Hash",  
 "dataType":"String",  
 "precision":3  
 },  
 {  
 "kind":"Hash",  
 "dataType":"Number",  
 "precision":7  
 }  
 ]  
 }  
 ],  
 "ExcludedPaths":[  
 {  
 "path":"/nonIndexedContent/\*"  
 }  
 ]  
}

## Next Steps

Follow the links below for index policy management samples and to learn more about DocumentDB’s query language.

1. [DocumentDB .NET Index Management code samples](https://github.com/Azure/azure-documentdb-net/blob/master/samples/code-samples/IndexManagement/Program.cs)
2. [DocumentDB REST API Collection Operations](https://msdn.microsoft.com/library/azure/dn782195.aspx)
3. [Query with DocumentDB SQL](/documentation/articles/documentdb-sql-query)