

✓ 100 XP

Azure Cosmos DB for MongoDB indexes

3 minutes

Indexes are data structures that allow queries faster access to your documents. Azure Cosmos DB for MongoDB takes advantage of the core index-management capabilities of Azure Cosmos DB. In this unit, we'll introduce different types of indexes and how to set them up from the Azure portal Data Explorer or programmatically using collection commands. Indexes are set at the collection level.

By default, Azure Cosmos DB for MongoDB automatically indexes the `_id` field. Uniqueness is automatically enforced by the `_id` field per shard key.

Let's review the different types of indexes supported by the Azure Cosmos DB for MongoDB.

Index types

Single field indexes

As the name implies, single field indexes are those created on a single field. For single field indexes, the order (ascending/descending) doesn't matter. A query can use multiple single field indexes when available. There's a limit of 500 single field indexes per collection.

Let's look at a couple of ways to create a single field index. Let's assume that on the database *inventory* we'll have a collection named *products*, and we want to create a single field index for the column *name* and a separate single field index for the column *quantity*.

Create a single field index using the Azure portal Index Policy tab

Under your Azure Cosmos DB for MongoDB database in the Azure portal, select Data Explorer.

1. Expand *Inventory* then *products* and select **Settings**.
2. Select the **Indexing Policy** tab. Notice that by default you should already have two indexes pre-created, the `_id` single field index discussed earlier and the `***` wildcard index we'll discuss later. For now, ignore those indexes.
3. Under the first textbox, input **name** and select **Single Field** from the dropdown.
4. Under the next textbox, input **quantity** and select **Single Field** from the dropdown.
5. Select on the Save icon.

Azure Cosmos DB for MongoDB account

Search (Ctrl+/) Save Discard New Shell

Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Cost Management
Quick start
Notifications
Data Explorer

Settings

Connection String
Features
Replicate data globally
Default consistency
Backup & Restore
Firewall and virtual networks
Private Endpoint Connections
Data Migration
Advisor Recommendations
Identity

MONGO API

DATA

inventory

Scale

products

Documents

Settings 1

NOTEBOOKS

Notebooks is currently not available. We are working on it.

Settings x

Indexing Policy 2

You have not saved the latest changes made to your indexing policy. Please click save to confirm the changes.

For queries that filter on multiple properties, create multiple single field indexes instead of a compound index. Compound indexes are only used for sorting query results. If you need to add a compound index, you can create one using the Mongo shell.

Current index(es)

Definition	Type	Drop Index
_id	Single Field	
\$**	Wildcard	
3 name	Single Field	
4 quantity	Single Field	
	Select an index type	

Index(es) to be dropped

Create a single field index using the MongoDB Shell

Open Mongo Shell, make sure it's pointing to your Azure Cosmos DB for MongoDB account using the account's connection string.

Run the following commands to add the two indexes.

JavaScript

```
// change to the inventory database
use inventory
// add name, and quantity indexes to the products collection
db.products.createIndex({name:1})
db.products.createIndex({quantity:1})
// look at the list of indexes to see the two new indexes
db.products.getIndexes()
```

This *getIndexes* collection function should return a similar result as the one below:

JSON

```
[
  { v: 1, key: { _id: 1 }, name: '_id_', ns: 'inventory.products' },
  { v: 1, key: { '$**': 1 }, name: '$**_1', ns: 'inventory.products' },
  { v: 1, key: { name: 1 }, name: 'name_1', ns: 'inventory.products' },
  { v: 1, key: { quantity: 1 }, name: 'quantity_1', ns: 'inventory.products' }
]
```

Create a single field index programmatically

The following script will create a single field index.

Node.js

JavaScript

```
// connect to the database "inventory"
var InventoryDatabase = mongoClient.db("inventory");

// connect to the collection "products"
var productscollection = InventoryDatabase.collection('products');

// add name, and quantity indexes to the products collection
var resultn = await productscollection.createIndex({ name: 1 });
var resultq = await productscollection.createIndex({ quantity: 1 });
```

Compound indexes

Compound indexes are needed any time your queries need to sort on multiple fields at once. If you need to filter on multiple fields, then just create a single field index for each field in the filter since it saves on *indexing cost*. That said, if a compound index contains the filtering fields of a query, *filtering cost* would be the same if you use the compound index than if you use single field indexes for each field in the filter.

Compound indexes can't be created on nested properties or arrays.

Unlike single field indexes where the order of the index was irrelevant, compound index must match the same sequence of fields as the query. Suppose you create the following compound index sorting the *doctors* by first name and then last name.

JavaScript

```
db.doctors.createIndex({firstName:1, lastName:1})
```

Now suppose you want to return all doctors by last name and then first name.

JavaScript

```
db.doctors.find().sort({lastName:1, firstName:1})
```

The previous compound index wouldn't work, you'll need the following index.

JavaScript

```
db.doctors.createIndex({lastName:1,firstName:1})
```

One more interesting note about compound indexes, they can be used for reversed sorts too, so the previous index will also work to return all doctors by last name descending and then first name descending (notice the **-1** representing descending).

JavaScript

```
db.doctors.find().sort({lastName:-1,firstName:-1})
```

ⓘ Note

You can't create a compound index using the Azure portal.

Let's suppose we want to add a compound index by **lastName** descending and **firstName** descending for the *doctors* collection in the *employees* database.

Create a compound index using the MongoDB Shell

Open Mongo Shell, make sure it's pointing to your Azure Cosmos DB for MongoDB account using the account's connection string.

Run the following commands to add the two indexes.

JavaScript

```
// change to the employees database
use inventory
// add the lastName, firstName compound index to the doctors collection
db.doctors.createIndex({lastName:-1,firstName:-1})
// look at the list of indexes to see the new compound indexes
db.products.getIndexes()
```

This **getIndexes** collection function should return a similar result as the one below:

JSON

```
[
  { v: 1, key: { _id: 1 }, name: '_id_', ns: 'employees.doctors' },
  { v: 1, key: { '$**': 1 }, name: '$**_1', ns: 'employees.doctors' },
  { v: 1, key: { lastName: -1, firstName: -1 }, name: 'lastName_-1_first-
Name_-1',ns: 'employees.doctors' }
]
```

Create a compound index programmatically

The following script will create a single field index.

Node.js

JavaScript

```
// connect to the database "employees"
var EmployeeDatabase = mongoClient.db("employees");

// connect to the collection "doctors"
var doctorscollection = EmployeeDatabase.collection('doctors');

// add the lastName, firstName compound indexes to the doctors collection
var resultd = await doctorscollection.createIndex({lastName:-1,first-
Name:-1});
```

Wildcard indexes

One of the NoSQL properties is being schema-less. This means that you might not always know all fields in your documents beforehand. Wildcard indexes allow you to support queries with unknown fields or when your documents have many different possible properties. Let's review two documents below in a collection called *school*.

JSON

```
"Students":
[
  {
    "StudentName": "John Smith",
    "Mayor": "Undefined",
    "Classes":
    [
      {
        "ClassName": "History",
        "Level":100,
        "Teacher": "Mike Michael"
      },
      {
        "ClassName": "Chemistry",
        "Level": 101,
        "Teacher": "Yi Yan",
        "Lab": "Chem Lab 101"
      }
    ]
  }
]
```

JSON

```
"Students":
[
  {
    "FirstName": "John",
    "LastName": "Smith",
    "GPA": 3.85,
    "FavoriteClasses":
    [
      {
        "ClassName": "Music"
      },
      {
        "ClassName": "Math",
        "Level": 101
      }
    ]
  }
]
```

While both documents have the ***Students*** property, the subproperties are all different. So in this case we might want to index all the ***Students*** subproperties. To do so in Mongo Shell, you would run the following command.

JavaScript

```
db.school.createIndex({"Students.$**" : 1})
```

We actually saw a wildcard index earlier, the index `$**` in the screenshot at the top of this unit. The index `$**` wildcard index will index *all* fields in the collection. We recommend using this index at the start of your database app development, or at least until you know how your collection will be queried. However, if your documents have many fields, keeping this wildcard index might create a large number of RU charges for writes and updates. Consider using individual indexes instead of this composite index in write-heavy workload environments.

Index properties

Azure Cosmos DB for MongoDB supports several index properties depending on the version. We'll look a couple of them here, to learn more, please review the [supported indexes and indexed properties](#) article.

Unique indexes

To enforce uniqueness in your documents, you'll need to set the **unique** property to **true** when defining your index. Let's assume we want to make sure the *deviceId* is unique in our *IOT* collection. Using Mongo Shell we would run the following command to ensure uniqueness.

JavaScript

```
db.IOT.createIndex( { "deviceId" : 1 }, {unique:true} )
```

If our collection is sharded, we'll also need to provide the partition key to create the unique index. This means our unique index will actually end up being a compound index. Let's assume that our shard key is *locationId* in our previous example, to create the unique index we would run the following command.

JavaScript

```
db.IOT.createIndex( {"locationId": 1, "deviceId" : 1 }, {unique:true} )
```

ⓘ Note

You can only create a unique index on an empty collection.

Unique partial indexes

Azure Cosmos DB for MongoDB gives us the ability to create unique indexes on a subset of the documents in our collection that meet a criteria defined by the ***partialFilterExpression*** property. This will enforce uniqueness on any document that meets the Filter criteria preventing any duplicates from being inserted. However, this filter won't prevent any document that doesn't meet the criteria from also being inserted.

To create a partial index, use the *db.collection.createIndex()* method with the *partialFilterExpression* option and *unique* constraint. The *partialFilterExpression* option accepts a document that specifies the filter condition using:

- equality expressions (that is, field: value or using the \$eq operator)
- \$exists: true expression
- \$gt, \$gte, \$lt, \$lte expressions
- \$type expressions
- \$and operator at the top-level only

Let's revisit our *IOT* example above. Let's assume that we have several years of IOT log data in this collection. Since IOT devices wouldn't insert data in the past, we might only care to check

uniqueness in the current year. In Mongo Shell, we would run the following command to create this unique index.

JavaScript

```
db.IoT.createIndex( { "deviceId" : 1 }, {unique:true, partialFilterExpression:{logYear:{$gte:2022}}} )
```

Background index updates

All index updates are always done in the background. Index updates consume Request Units at a lower priority than other database operations, so they'll cause no downtime to your writes, updates or deletes.

Adding a new index doesn't affect read availability. While the index is being created Azure Cosmos DB will use the existing indexes to support queries. No inconsistency of query results will happen when new indexes are being built or added.

For more information on MongoDB indexes please review the [Manage indexing in Azure Cosmos DB for MongoDB](#) document.

Next unit: Aggregation pipeline

Continue >
