Lab 4 Troubleshooting and Tuning Cosmos DB Requests

In this lab, you will use the Python SDK to tune an Azure Cosmos DB request to optimize performance of your application.

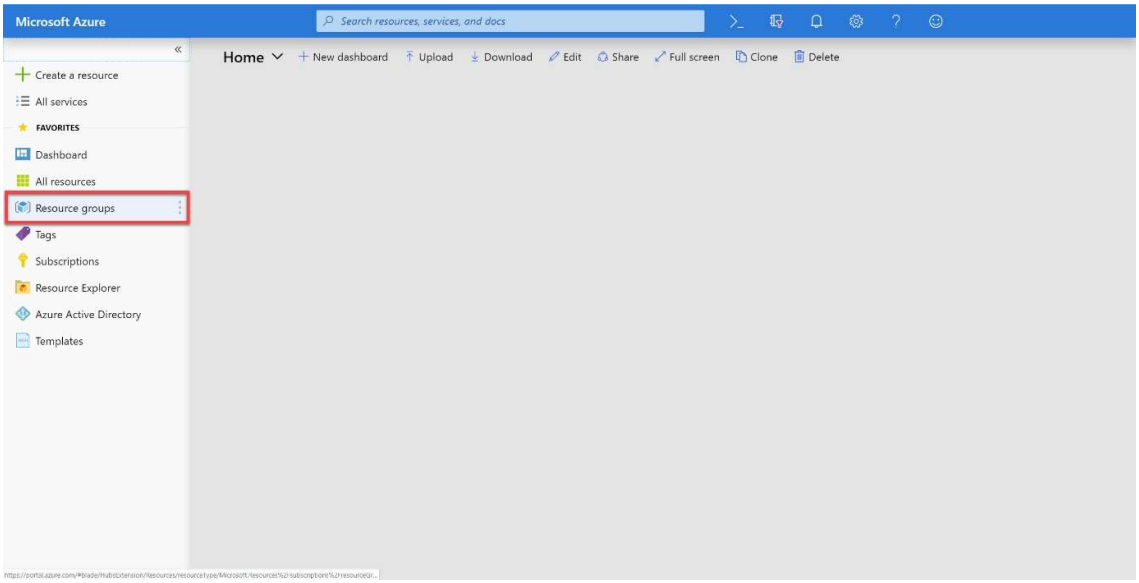
Setup

Before you start this lab, you will need to create an Azure Cosmos DB database and collection that you will use throughout the lab. You will also use the Data Migration Tool to import existing data into your collection.

Create Azure Cosmos DB Database and Collections

You will now create a database and collection within your Azure Cosmos DB account.

1. On the left side of the portal, click the Resource groups link.



2. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
3. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.
4. In the Azure Cosmos DB blade, locate and click the Overview link on the left side of the blade.
5. At the top of the Azure Cosmos DB blade, click the Add Collection button.

6. In the Add Collection popup, perform the following actions:
   a. In the Database id field, select the Create new option and enter the value FinancialDatabase.
   b. Ensure the Provision database throughput option is not selected.
   c. In the Collection id field, enter the value TransactionCollection.
   d. In the Storage capacity section, select the Unlimited option.
   e. In the Partition key field, enter the value `/costCenter`.
   f. In the Throughput field, enter the value `10000`.
   g. Click the OK button.
7. Wait for the creation of the new database and collection to finish before moving on with this lab.
8. We now need to created a fixed-sized collection, but this is no longer possible from the portal so we will make use of the python API, using a similar script to those we used in Lab 1 to create a fixed size collection in the Financial Database

9. Open-up Visual Studio Code and open the source directory you have been working in.
   a) Create a new file called createFixedCollLab3.py
   b) Copy and paste the code from the python script below into the new file.

   c) Replace the line:
   ```
   databaseID = 'DatabaseName'
   ```
   with
   ```
   databaseID = 'FinancialDatabase'
   ```

   d) Replace the line:
   ```
   collectionID = 'collectionName'
   ```
   with
   ```
   collectionID = 'PeopleCollection'
   ```

   e) Replace the throughput attribute in the offer directory:
   ```
   'offerThroughput': 400
   ```
   with
   ```
   'offerThroughput': 1000
   ```

   f) Save the modified python file with Ctrl-S
   g) Run the script by pressing the F5 key
10. The script should complete without errors and should give something similar the following output

```
Database self link is: dbs/ShUOAA==/
Collection self link for PeopleCollection is: dbs/ShUOAA==/colls/ShUOAK9tS+U=/
```

```python
import azure.cosmos as cosmos
import azure.cosmos.cosmos_client as cosmos_client
import json

with open('config.json', 'r') as f:
    config = json.load(f)

client = cosmos_client.CosmosClient(config['endPoint'],{'masterKey': config['key']})
databaseID =   'DatabaseName'
collectionID = 'CollectionName'

try:
    db = client.CreateDatabase({ 'id': databaseID} )
except Exception as ex:
    if type(ex) == cosmos.errors.HTTPFailure and ex.status_code == 409:
        # database already exists
        db = client.ReadDatabase('dbs/' + databaseID )
    else:
        print("we had a problem creating database")
        raise ex

print('Database self link is: {}'.format(db['_self']))

options = {
                'offerEnableRUPerMinuteThroughput': True,
                'offerVersion': "V2",
                'offerThroughput': 400
}

try:
    collection = client.CreateContainer(db['_self'], { 'id': collectionID }, options)
except Exception as ex:
    if type(ex) == cosmos.errors.HTTPFailure and ex.status_code == 409:
        # collection already exists
        collection = client.ReadContainer('dbs/' + databaseID + '/colls/' + collectionID )
    else:
        print("we had a problem creating collection")
        raise ex

print('Collection self link for {} is: {}'.format(collectionID, collection['_self']))
```

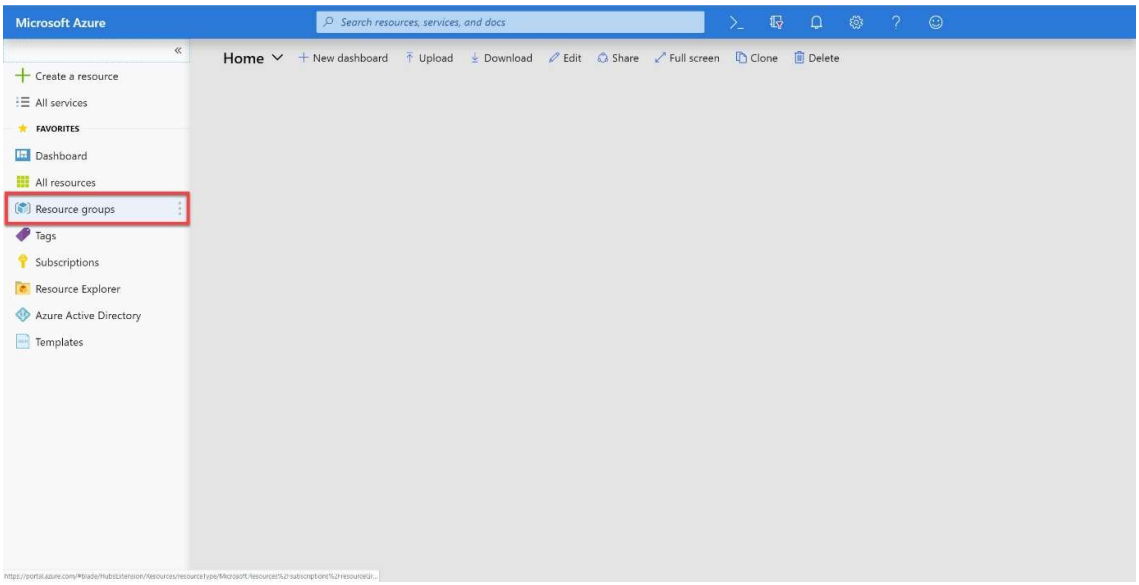
createFixedCollLab3.py Script

Import Lab Data into Collection

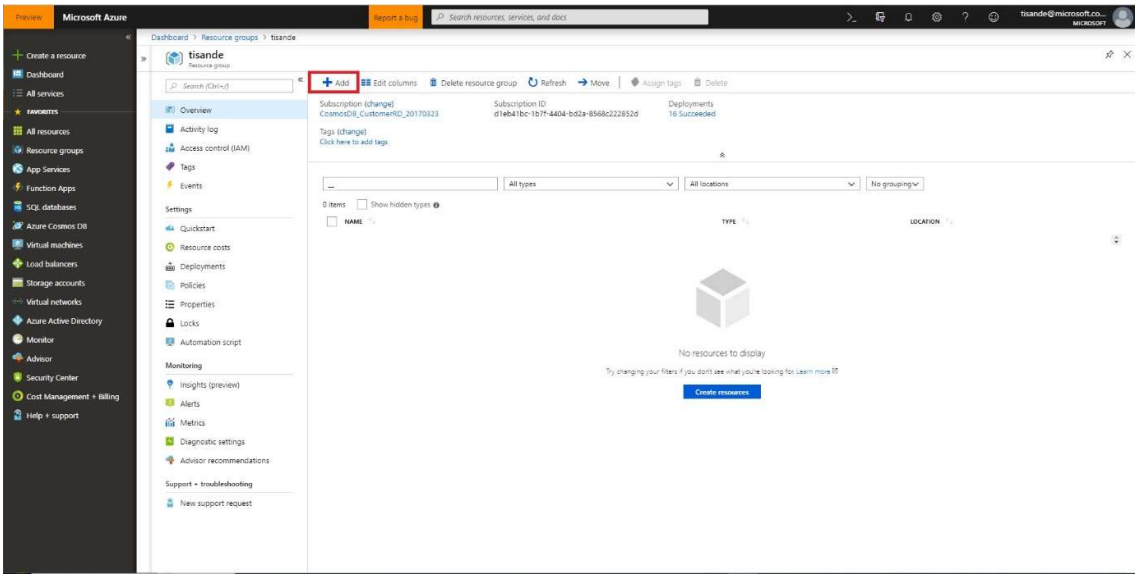
You will use Azure Data Factory (ADF) to import the JSON array stored in the students.json file from Azure Blob Storage.

1. On the left side of the portal, click the Resource groups link.

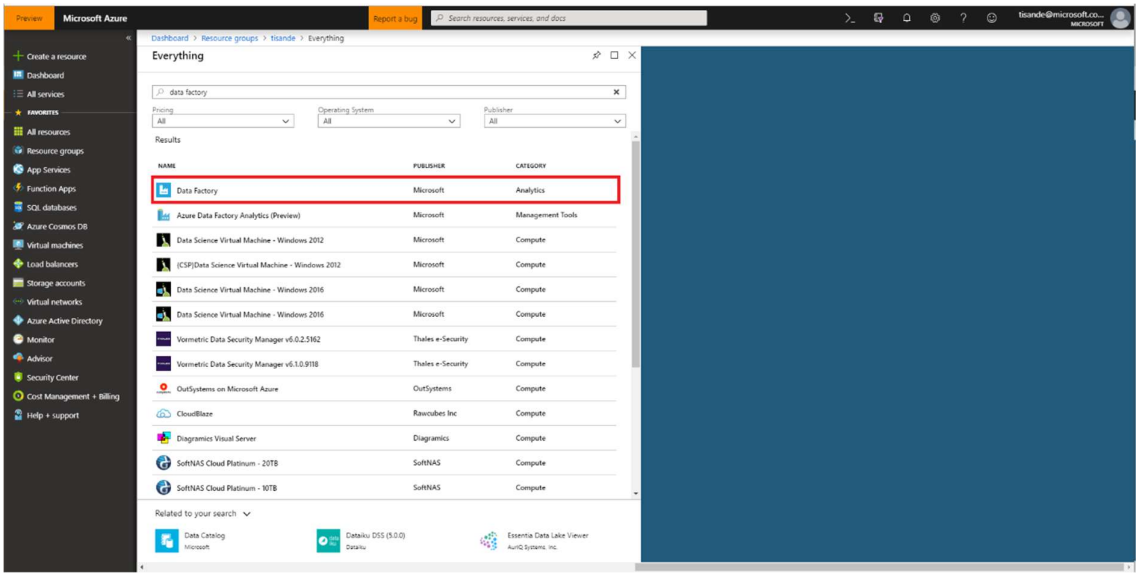
> To learn more about copying data to Cosmos DB with ADF, please read ADF's documentation.



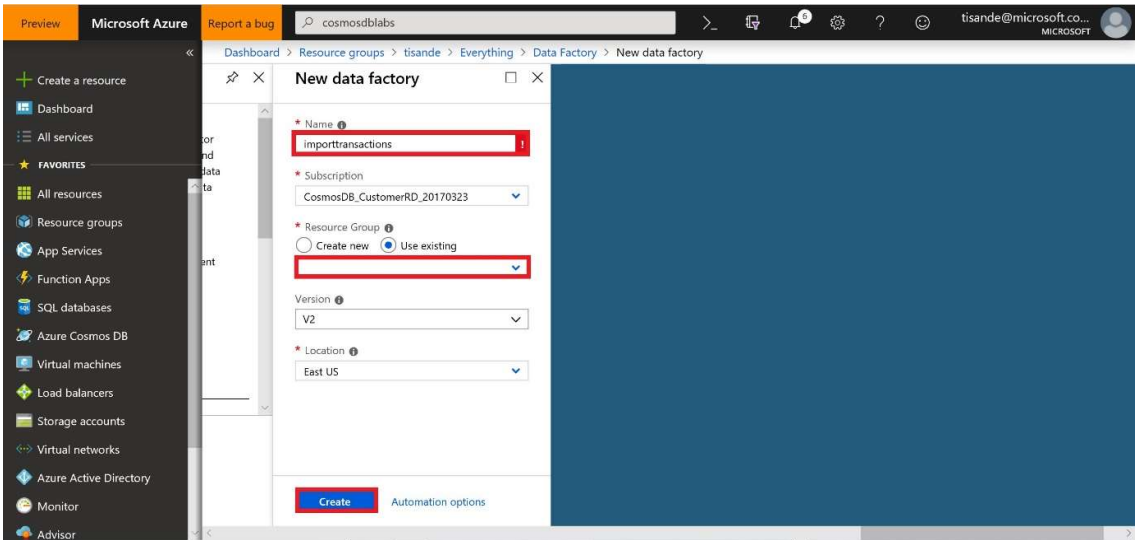
2. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
3. Click add to add a new resource



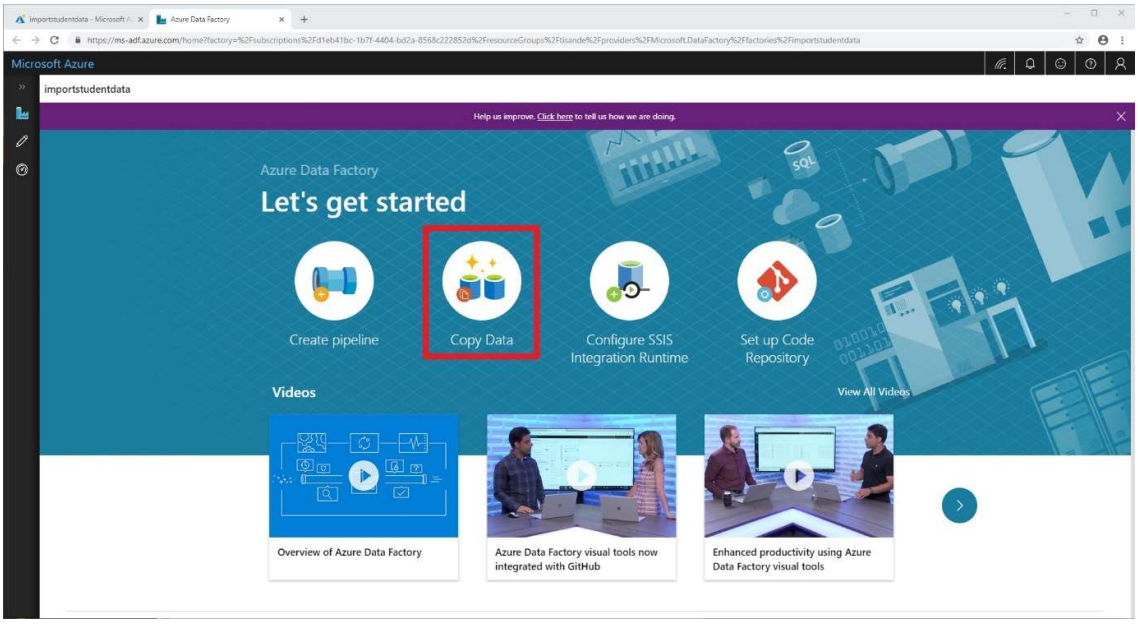
4. Search for Data Factory and select it



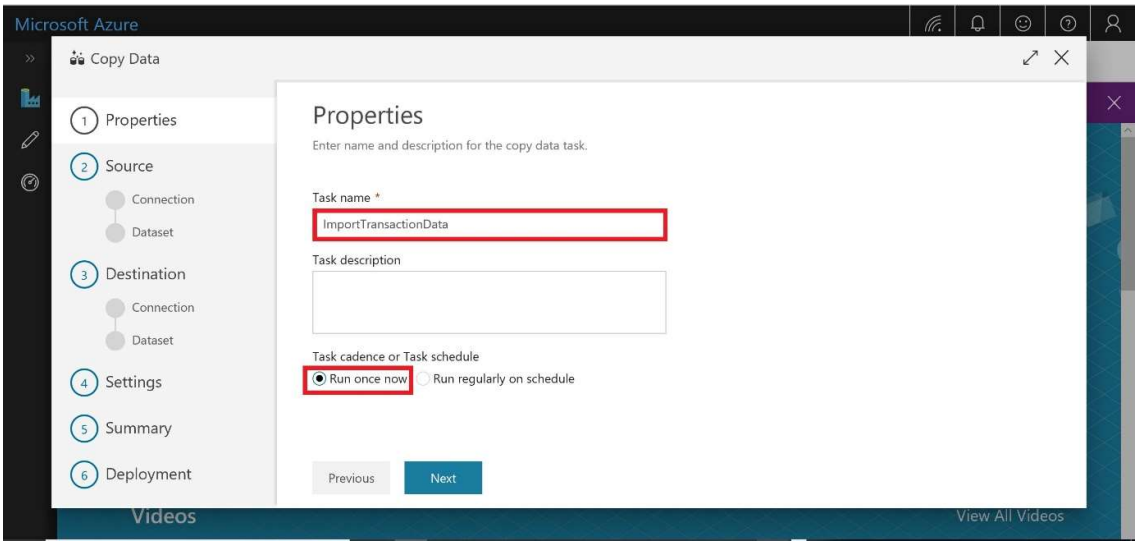
5. Create a new Data Factory. You should name this data factory importtransactionsXX (where XX is your initials) and select the relevant Azure subscription. You should ensure your existing cosmosdblab-group resource group is selected as well as a Version V2.
Select North Europe as the region. Click create.



6. Select Copy Data. We will be using ADF for a one-time copy of data from a source JSON file on Azure Blob Storage to a database in Cosmos DB's SQL API. ADF can also be used for more frequent data transfer from Cosmos DB to other data stores.



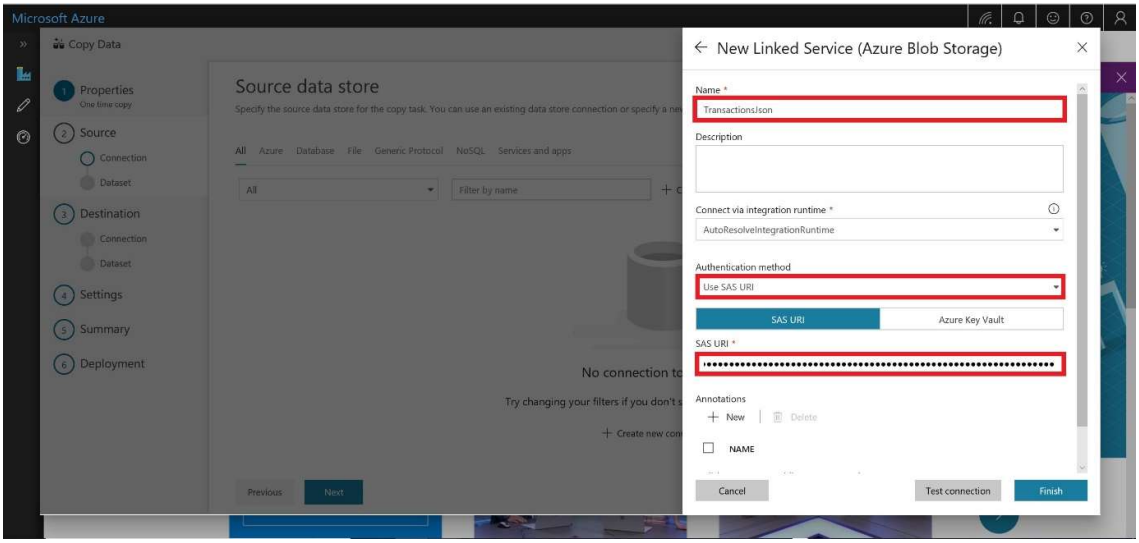
7. Edit basic properties for this data copy. You should name the task ImportTransactionsData and select to Run once now.



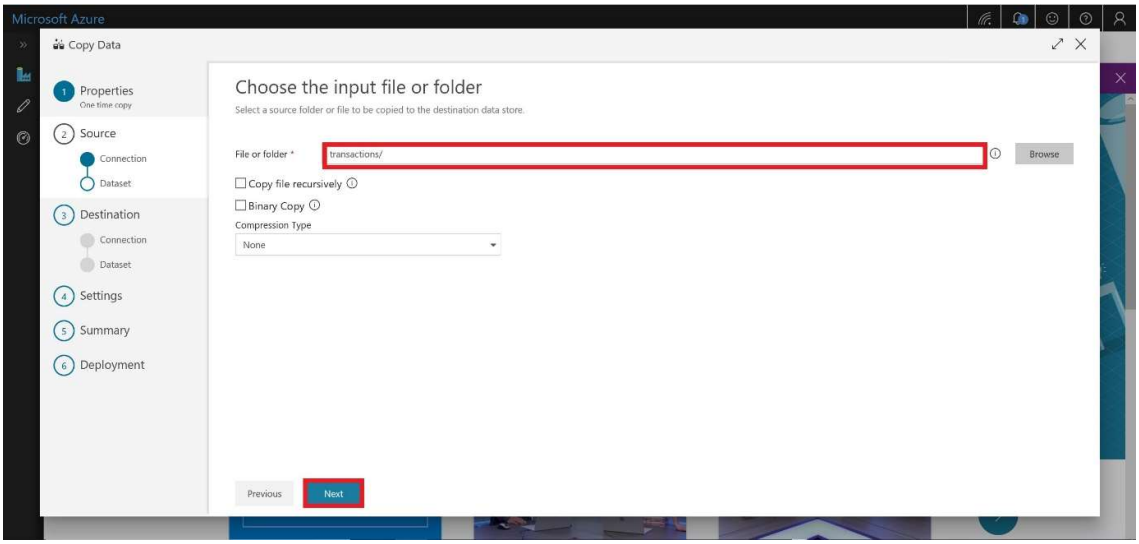
8. Create a new connection and select Azure Blob Storage. We will import data from a json file on Azure Blob Storage. In addition to Blob Storage, you can use ADF to migrate from a wide variety of sources. We will not cover migration from these sources in this tutorial.

9. Name the source TransactionsJson and select Use SAS URI as the Authentication method. Please use the following SAS URI for read-only access to this Blob Storage container:

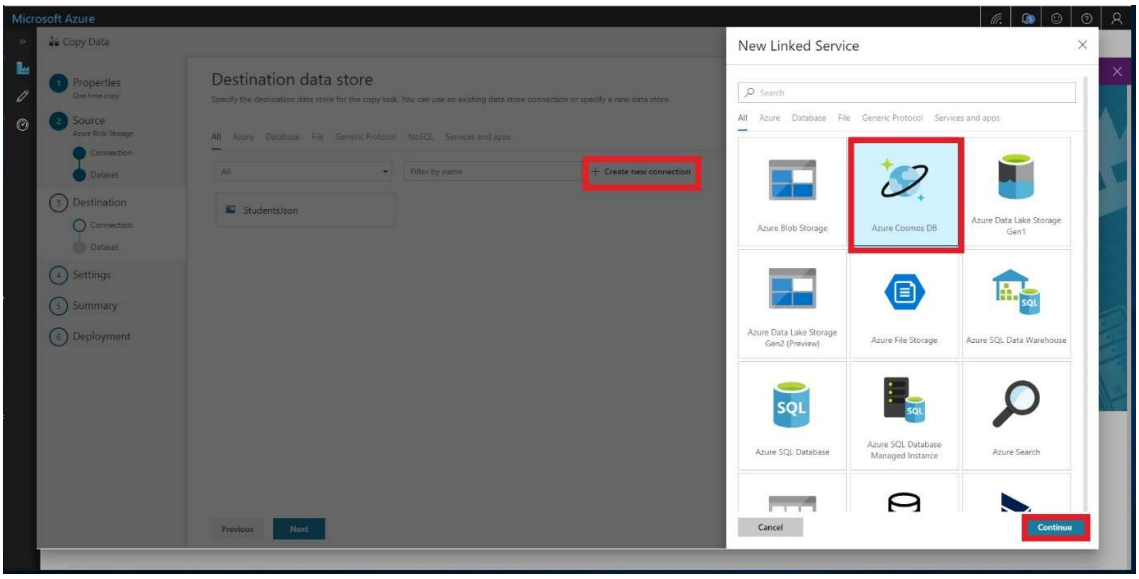
   https://cosmosdblabs.blob.core.windows.net/?sv=2017-11-09&ss=bfqt&srt=sco&sp=rwdlacup&se=2020-03-11T08:08:39Z&st=2018-11-10T02:08:39Z&spr=https&sig=ZSwZhcBdwLVIMRj94pxxGojWwyHkLTAgnL43BkbWKyg%3D



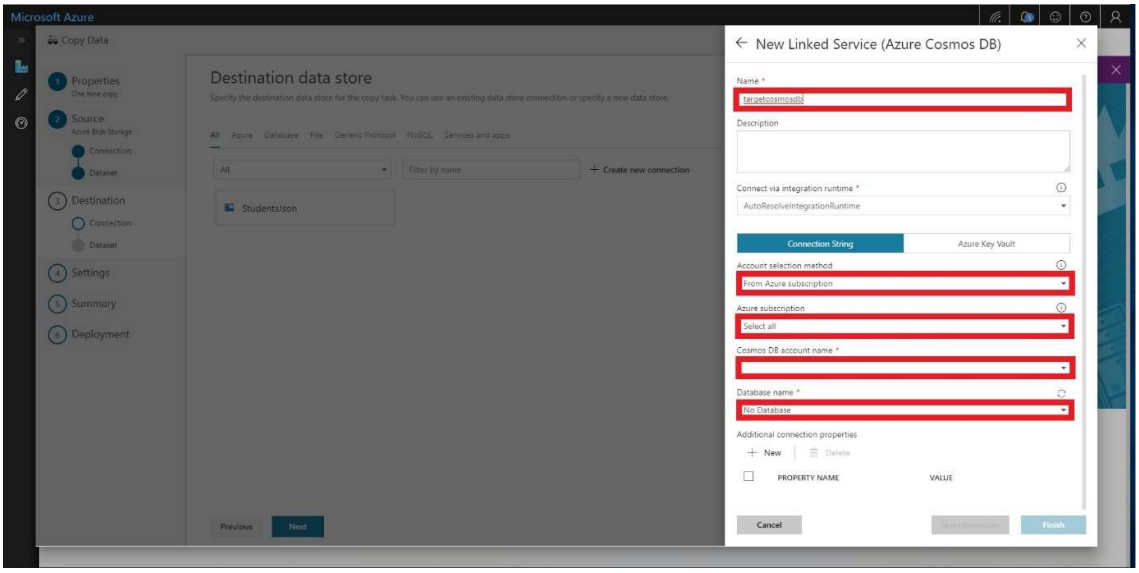
10. Select the transactions folder.



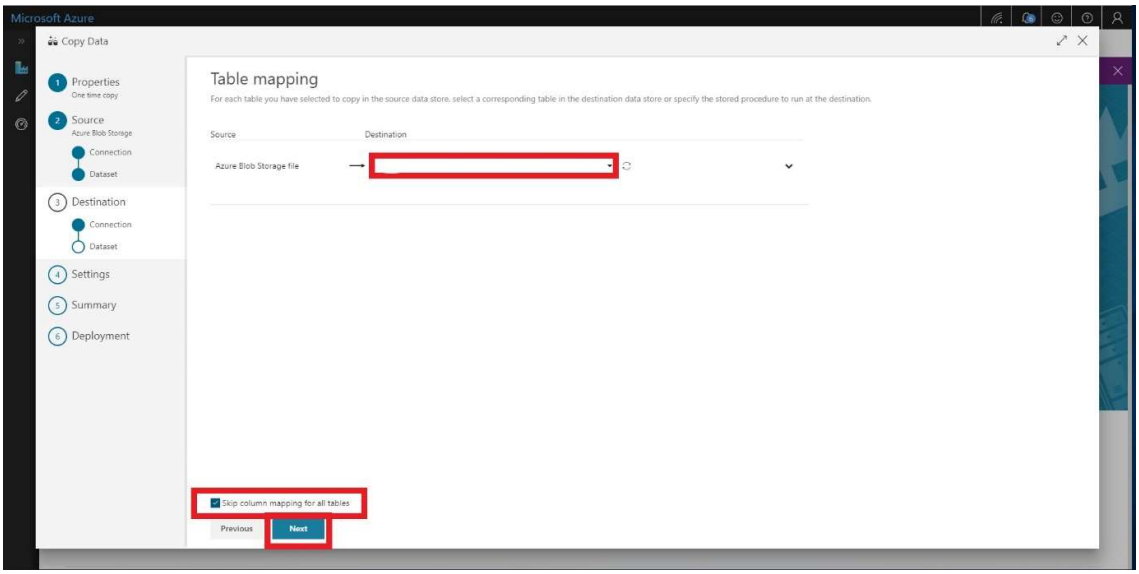
11. Ensure that Copy file recursively and Binary Copy are not checked off. Also ensure that Compression Type is “none”.
12. ADF should auto-detect the file format to be JSON. You can also select the file format as JSON format. You should also make sure you select Array of Objects as the File pattern
13. You have now successfully connected the Blob Storage container with the transactions.json file. You should select **TransactionsJson**as the source and click **Next**
14. Add the Cosmos DB target data store by selecting Create new connection and selecting Azure Cosmos DB



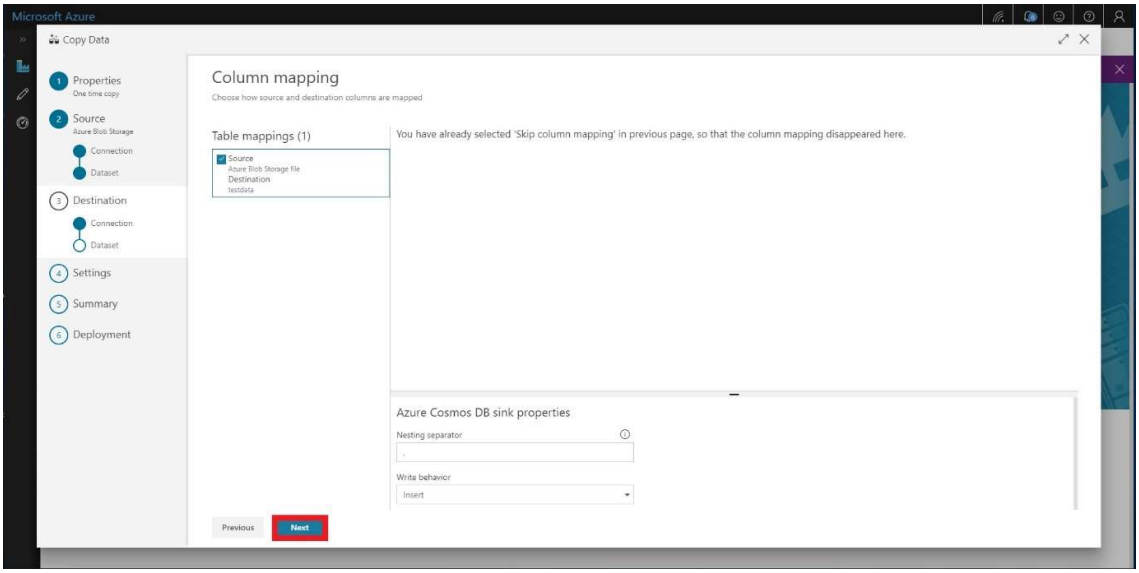
15. Name the linked service targetcosmosdb and select your Azure subscription and Cosmos DB account. You should also select the Cosmos DB database (FinancialDatabase) that you created earlier.



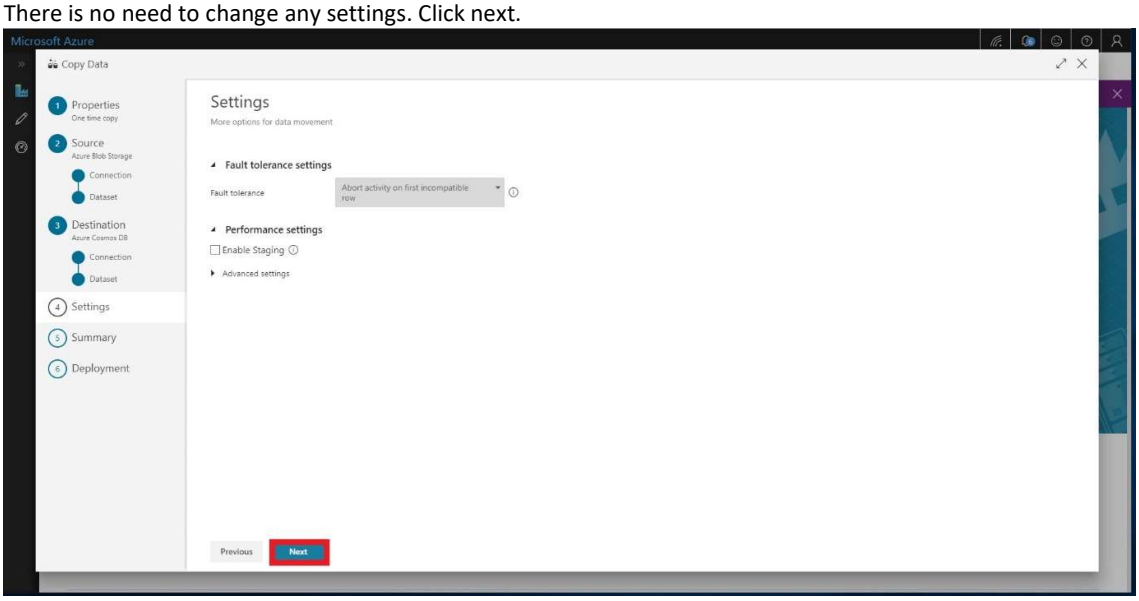
16. Select your newly created targetcosmosdb connection as the Destination date store.
17. Select your collection (TransactionCollection) from the drop-down menu. You will map your Blob storage file to the correct Cosmos DB collection.



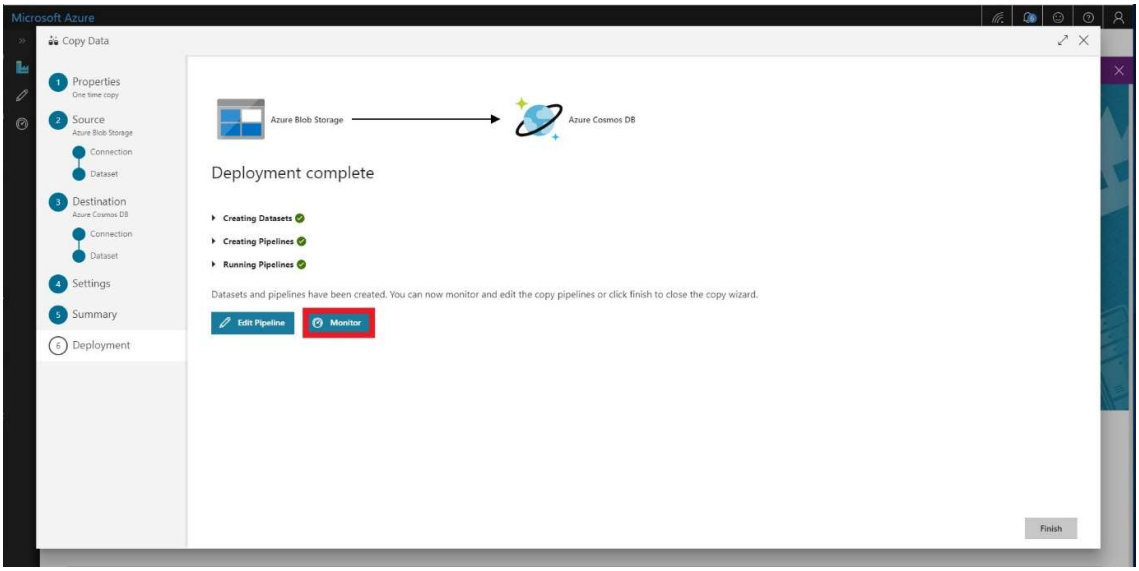
18. You should have selected to skip column mappings in a previous step. Click through this screen.



19. There is no need to change any settings. Click next.



20. After Deployment is complete. Click Monitor to see the progress of the Data Factory job.



21. After a few minutes, refresh the page and the status for the ImportTransactions pipeline should be listed as Succeeded.
22. Once the import process has completed, close the ADF. You will now proceed to execute simple queries on your imported data.

Using Response Headers to Optimise Queries and Inserts

Observe RU Charge for Large Document

In this part of the exercise we will investigate how we can judge the cost of our interactions with Cosmos DB and learn about the measures we can take to ensure that we are using our database resources efficiently. We will start by creating a simple Python function to create a fake person which we will insert into our PeopleCollection in the FinancialDatabase database. The code is shown below.

```
from faker import Faker

def create_person():
    myfactory = Faker('en_GB')
    firstName = myfactory.first_name()
    lastName = myfactory.last_name()
    fullName = firstName + " " + lastName
    address = myfactory.address()
    phoneNumber = myfactory.phone_number()
    email = myfactory.email()
    job = myfactory.job()
    company = myfactory.company() + " " + myfactory.company_suffix()
    geo = myfactory.local_latlng(country_code="US", coords_only=False)
    return {
        'FirstName': firstName,
        'LastName':  lastName,
        'FullName': fullName,
        'Address': address,
        'PhoneNumber': phoneNumber,
        "EmailAddress" : email,
        "Profession" : job,
        "Company" : company,
        "CompanyAddress" : myfactory.address(),
        "HQLocation" : geo

    }

if __name__ == "__main__":
    print(create_person() )
```

fake_person.py

The create_person() function is quite simple and uses the faker module to randomly create a fictitious person and return it as a dictionary object. The script above will run in your python environment in Visual Studio Code and it is worth executing to see the output of the function.

We will use the create_person() function to create some documents and insert them into the PeopleCollection on the next page.

1. Create a new file in Visual Studio Code named cr_person.py and copy in the code from below

```python
import azure.cosmos as cosmos
import azure.cosmos.cosmos_client as cosmos_client
import json
from faker import Faker

def create_person():
    myfactory = Faker('en_GB')
    firstName = myfactory.first_name()
    lastName = myfactory.last_name()
    fullName = firstName + " " + lastName
    address = myfactory.address()
    phoneNumber = myfactory.phone_number()
    email = myfactory.email()
    job = myfactory.job()
    company = myfactory.company() + " " + myfactory.company_suffix()
    geo = myfactory.local_latlng(country_code="US", coords_only=False)
    return {
        'FirstName': firstName,
        'LastName':  lastName,
        'FullName': fullName,
        'Address': address,
        'PhoneNumber': phoneNumber,
        "EmailAddress" : email,
        "Profession" : job,
        "Company" : company,
        "CompanyAddress" : myfactory.address(),
        "HQLocation" : geo
    }

if __name__ == "__main__":
    # get the access key and endpoint and set up the client
    with open('config.json', 'r') as f:
        config = json.load(f)

    # create a client and set up the database and collection links
    client = cosmos_client.CosmosClient(config['endPoint'],
                                {'masterKey':    config['key']})

    databaseID = "FinancialDatabase";
    collectionID = "PeopleCollection"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID

    personDoc = create_person()

    personResp = client.CreateItem(collection_link, personDoc)

    # Get the resource cost from response headers
    responseHeaders = client.last_response_headers
    resourceUsage = responseHeaders['x-ms-request-charge']

    if personResp:
        print("Document created - Resource cost was {} RU's".format(resourceUsage))
    else:
        print("No document created")
```

cr_person.py

2. Notice how the script creates a CosmosDB client and then creates a person document or dictionary and then loads it using the client.CreateItem call. The script then gets the HTTP response headers to determine the resource cost for the call. Press F5 to run the script and record the resource cost.

3. Return to the Azure Portal (http://portal.azure.com).
4. On the left side of the portal, click the Resource groups link.
5. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
6. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.
7. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.
8. In the Data Explorer section, expand the FinancialDatabase database node and then observe select the PeopleCollection node.
9. Click the New SQL Query button at the top of the Data Explorer section.
10. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT TOP 2 * FROM coll ORDER BY coll._ts DESC
```

This query will return the latest two documents added to your collection.

11. Click the Execute Query button in the query tab to run the query.
12. In the Results pane, observe the results of your query.

13. Copy the cr_person.py file to a new file named cr_family.py and replace the main body of the script with the following code :

```
if __name__ == "__main__":
    # get the access key and endpoint and set up the client
    with open('config.json', 'r') as f:
        config = json.load(f)

    # create a client and set up the database and collection links
    client = cosmos_client.CosmosClient(config['endPoint'],
                                {'masterKey':    config['key']})

    databaseID = "FinancialDatabase";
    collectionID = "PeopleCollection"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID

    familyDoc = { "Person" : create_person(),
                  "Relatives" : {
                      "Spouse" : create_person(),
                      "Children" : [create_person(),
                                    create_person(),
                                    create_person(),
                                    create_person()]
                  }
                }

    clientResp = client.CreateItem(collection_link, familyDoc)

    # Get the resource cost from response headers
    responseHeaders = client.last_response_headers
    resourceUsage = responseHeaders['x-ms-request-charge']

    if clientResp:
        print("Document created - Resource cost was {} RU's".format(resourceUsage))
    else:
        print("No document created")
```

cr_family.py(main body)

14. The cr_family.py script creates a much larger family document consisting of a person, their spouse and 4 children and all of their sub attributes. Notice how when it is inserted into the PeopleCollection it requires many more resource units.

15. Return to the Azure Portal (http://portal.azure.com).
16. On the left side of the portal, click the Resource groups link.
17. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
18. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.
19. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.
20. In the Data Explorer section, expand the FinancialDatabase database node and then observe select the PeopleCollection node.
21. Click the New SQL Query button at the top of the Data Explorer section.
22. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM coll WHERE IS_DEFINED(coll.Relatives)
```

This query will return the only document in your collection with a property named Children.

23. Click the Execute Query button in the query tab to run the query.
24. In the Results pane, observe the results of your query.

Tuning the Index Policy to Reduce Resource Consumption

1. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.
2. In the Data Explorer section, expand the FinancialDatabase database node, expand the PeopleCollection node, and then select the Scale & Settings option.
3. In the Settings section, locate the Indexing Policy field and observe the current default indexing policy:

```
{
     "indexingMode": "consistent",
     "automatic": true,
     "includedPaths": [
         {
             "path": "/*",
             "indexes": [
                 {
                     "kind": "Range",
                     "dataType": "Number",
                     "precision": -1
                 },
                 {
                     "kind": "Range",
                     "dataType": "String",
                     "precision": -1
                 },
                 {
                     "kind": "Spatial",
                     "dataType": "Point"
                 }
             ]
         }
     ],
     "excludedPaths": []
 }
```

This policy will index all paths in your JSON document. This policy implements maximum percision (-1) for both numbers (max 8) and strings (max 100) paths. This policy will also index spatial data.

4. Replace the indexing policy with a new policy that removes the /Relatives/* path from the index:

```
{
     "indexingMode": "consistent",
     "automatic": true,
     "includedPaths": [
         {
             "path":"/*",
             "indexes":[
                 {
                     "kind": "Range",
                     "dataType": "String",
                     "precision": -1
                 },
                 {
                     "kind": "Range",
                     "dataType": "Number",
                     "precision": -1
                 }
             ]
         }
     ],
     "excludedPaths": [
         {
             "path":"/Relatives/*"
         },
         {
             "path": "/\"_etag\"/?"
         }
     ]
 }
```

This new policy will exclude the `/Relatives/*` path from indexing effectively removing the Children property of your large JSON document from the index.

5. Click the Save button at the top of the section to persist your new indexing policy and "kick off" a transformation of the collection's index.
6. Click the New SQL Query button at the top of the Data Explorer section.
7. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM coll WHERE IS_DEFINED(coll.Relatives)
```

8. Click the Execute Query button in the query tab to run the query.

You will see immediately that you can still determine if the /Relatives path is defined.

9. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM coll WHERE IS_DEFINED(coll.Relatives) ORDER BY
coll.Relatives.Spouse.FirstName
```

10. Click the Execute Query button in the query tab to run the query.

> This query will fail immediately since this property is not indexed.

11. Return to the Visual Studio Code window with the cr_family.py program loaded. Re-run the program and note that the resource usage for the insert has dropped. This is due to the document attributes under /Relatives no longer being indexed.

> Maintaining the indexes for complex documents accounts for a considerable amount of the cost for inserting documents in Cosmos DB as it will index almost all attributes by default

12. Return to the Azure Portal (http://portal.azure.com).
13. On the left side of the portal, click the Resource groups link.
14. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
15. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.
16. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.
17. In the Data Explorer section, expand the FinancialDatabase database node, expand the PeopleCollection node, and then select the Scale & Settings option
18. In the index policy remove the following element that excludes the /Relatives attributes :

```
{
   "path": "/Relatives/*"
},
```

19. Press Save to save the change and begin the online index rebuild
20. Return to the Query window with the query below in it :

```
SELECT * FROM coll WHERE IS_DEFINED(coll.Relatives) ORDER BY
coll.Relatives.Spouse.FirstName
```

21. Re-run the query and note that it completes this time now the /Relatives/* attributes are indexed now.

Implement Upsert using Response Status Codes

1. Create a new file in Visual Studio Code named uriWriteTest.py and copy the following python code into your script. The script creates a simple document in a dictionary called doc and sets up a document link using the Cosmos DB resource convention.
   See https://docs.microsoft.com/en-us/rest/api/cosmos-db/cosmosdb-resource-uri-syntax-for-rest for more details.
   The program then tries to read the document from the PeopleCollection, but it will fail as the document is not present in the database and the exception will be caught in the except cosmos.errors.HTTPFailure block

```
import azure.cosmos as cosmos
import azure.cosmos.cosmos_client as cosmos_client
import json

if __name__ == "__main__":
    # get the access key and endpoint and set up the client
    with open('config.json', 'r') as f:
        config = json.load(f)

    # create a client and set up the database and collection links
    client = cosmos_client.CosmosClient(config['endPoint'],
                                {'masterKey':    config['key']})

    databaseID = "FinancialDatabase";
    collectionID = "PeopleCollection"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID

    document_id = "example.document"
    document_link = collection_link + '/docs/' + document_id

    doc = { "id": "example.document",
            "FirstName": "Example",
            "LastName": "Person"}

    try:
        read_response = client.ReadItem(document_link)
        print("ok doc found")
    except cosmos.errors.HTTPFailure as ex:
        print("oops not found " + str(ex.status_code) + " error "  )

    print("done !")
```

uriWrite.py

2. Run the uriWrite.py script in Visual Studio Code by pressing F5 – note the exception is caught and a 404 error was returned to the client.ReadItem call.
3. Copy the uriWrite.py script and rename it to upsertWrite.py
4. Delete the try: except block below

```
    try:
        read_response = client.ReadItem(document_link)
        print("ok doc found")
    except cosmos.errors.HTTPFailure as ex:
        print("oops not found " + str(ex.status_code) + " error "  )
```

With the code on the next page

```
 read_response = client.UpsertItem(collection_link, doc)
if read_response:
    print("example.document loaded ok"   )

print("done !")
```

5. Run the code by pressing F5 in Visual Studio Code

6. Return to the Azure Portal (http://portal.azure.com).
7. On the left side of the portal, click the Resource groups link.
8. In the Resource groups blade, locate and select the cosmosgroup-lab Resource Group.
9. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.
10. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.
11. In the Data Explorer section, expand the FinancialDatabase database node and then observe select the PeopleCollection node.
12. Click the New SQL Query button at the top of the Data Explorer section.
13. In the query tab, replace the contents of the query editor with the following SQL query:

```
SELECT * FROM coll WHERE coll.id = “example.document”
```

14. Verify that the document is present.
15. Modify the document definition for the doc variable and re-run the script
16. Re-run the query above in the Azure Portal Cosmos DB Data Explorer panel to verify that your changes have made it into the database collection.

Troubleshooting Requests

First, you will use the Python SDK to issue request beyond the assigned capacity for a container. Request unit consumption is evaluated at a per-second rate. For applications that exceed the provisioned request unit rate, requests are rate-limited until the rate drops below the provisioned throughput level. When a request is rate-limited, the server pre-emptively ends the request with an HTTP status code of `429 RequestRateTooLargeException` *and returns the* `x-ms-retry-after-ms` *header. The header indicates the amount of time, in milliseconds, that the client must wait before retrying the request. You will observe the rate-limiting of your requests in an example application.*

Reducing R/U Throughput for a Collection

1. Return to the Azure Portal (http://portal.azure.com).
   On the left side of the portal, click the Resource groups link.
2. In the Resource groups blade, locate and select the cosmosgroup-lab *Resource Group*.
3. In the cosmosgroup-lab blade, select the Azure Cosmos DB account you recently created.
4. In the Azure Cosmos DB blade, locate and click the Data Explorer link on the left side of the blade.
5. In the Data Explorer section, expand the FinancialDatabase database node, expand the TransactionCollection node, and then select the Scale & Settings option.
6. In the Settings section, locate the Throughput field and update it's value to 400.

   This is the minimum throughput that you can allocate to an *unlimited* collection.

7. Click the Save button at the top of the section to persist your new throughput allocation.

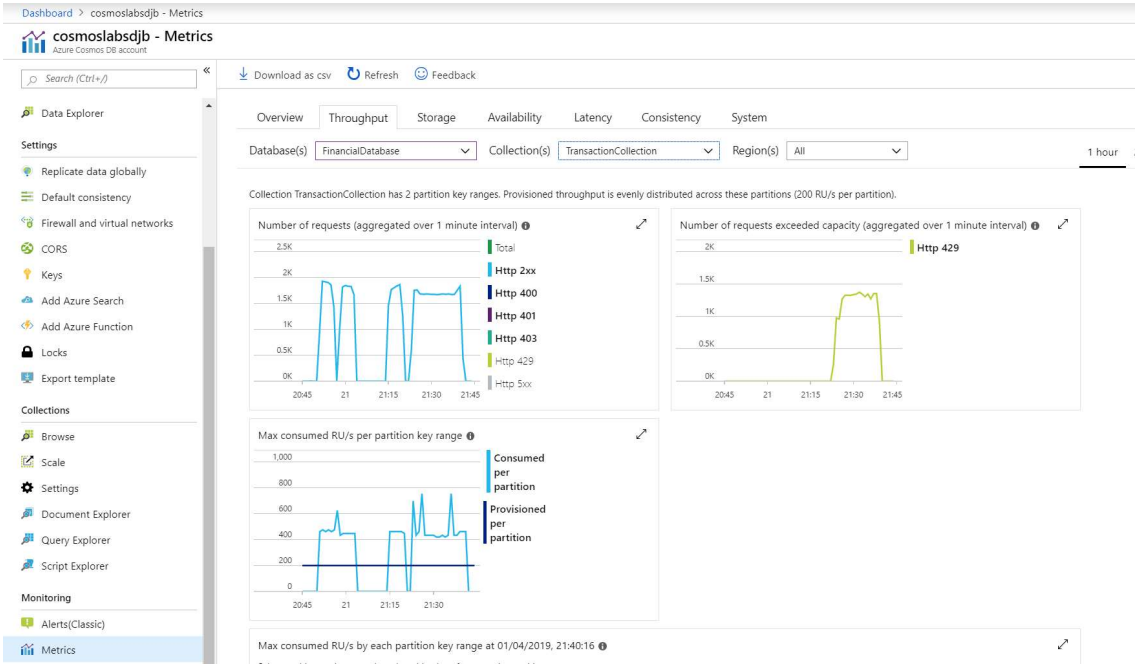
Observing Throttling (HTTP 429)

The following script will create 10,000 transaction documents in a list structure stored in the memory of your computer. It will then proceed to load these documents in a tight loop into the FinancialDatabase with the rate limited to 400 RU's. Depending on the speed of your connection to the Azure Cosmos DB service – this should be enough to exceed 400 RU's and to experience throttling.

1. Create a new python source file in Visual Studio Code called TxLoadGen.py
2. Copy the code overleaf into the new file
3. Save and execute the TxLoadGen.py by pressing F5
4. Once the list of 10,000 documents has been created, observe the amount of resource being consumed in the Azure Portal Cosmos DB -> Metrics -> Throughput panel

```python
import azure.cosmos as cosmos
import azure.cosmos.cosmos_client as cosmos_client
import json
import random
from faker import Faker

DOCS_TO_CREATE = 10000

def create_tx():
    """Return a fake tx object - create factory myfactory outside call"""
    firstName = myfactory.first_name()
    lastName = myfactory.last_name()
    paidBy = firstName.lower() + lastName.lower()
    processed = myfactory.boolean(chance_of_getting_true=80)
    amount = myfactory.pydecimal(left_digits=4, right_digits=2, positive=True)
    costCentres = ['toys','computers', 'jewelery','automotive','outdoors','sports',
                   'garden', 'books', 'tools', 'grocery', 'industrial']
    costCenter = costCentres[random.randint(0,10)]

    return {
        'amount': float(amount),
        'paidBy': paidBy,
        'processed':  processed,
        'costCenter': costCenter
    }

if __name__ == "__main__":
    myfactory = Faker('en_GB')
    txList = []

    print("Creating a list of 10,000 documents")
    for cnt in range(DOCS_TO_CREATE):
        txList.append(create_tx())
        if cnt%100 == 0 :
            print('.', end="")
    print('\nCreated a list of 10,000 documents.')

    # get the access key and endpoint and set up the client
    with open('config.json', 'r') as f:
        config = json.load(f)

    # create a client and set up the database and collection links
    client = cosmos_client.CosmosClient(config['endPoint'],
                                {'masterKey':    config['key']})

    databaseID = "FinancialDatabase";
    collectionID = "TransactionCollection"
    database_link = 'dbs/' + databaseID
    collection_link = database_link + '/colls/' + collectionID

    for cnt in range(DOCS_TO_CREATE):
        TxResp = client.CreateItem(collection_link, txList[cnt])
        if cnt%100 == 0 :
            print('.', end="")

    print('\nCreated 10,000 documents in the TransactionCollection')
```

TxLoadGen.py


Dashboard > cosmoslabsdjb - Metrics
cosmoslabsdjb - Metrics
Azure Cosmos DB account
Search (Ctrl+/)
Data Explorer
Settings
Replicate data globally
Default consistency
Firewall and virtual networks
CORS
Keys
Add Azure Search
Add Azure Function
Locks
Export template
Collections
Browse
Scale
Settings
Document Explorer
Query Explorer
Script Explorer
Monitoring
Alerts(Classic)
Metrics
Download as csv
Refresh
Feedback
Overview
Throughput
Storage
Availability
Latency
Consistency
System
Database(s) FinancialDatabase
Collection(s) TransactionCollection
Region(s) All
1 hour
Collection TransactionCollection has 2 partition key ranges. Provisioned throughput is evenly distributed across these partitions (200 RU/s per partition).
Number of requests (aggregated over 1 minute interval)
Total
Http 2xx
Http 400
Http 401
Http 403
Http 429
Http 5xx
Number of requests exceeded capacity (aggregated over 1 minute interval)
Http 429
Max consumed RU/s per partition key range
Consumed per partition
Provisioned per partition
Max consumed RU/s by each partition key range at 01/04/2019, 21:40:16