Atlas Data Federation Cookbook: Recipes for Data Engineering

Author: John Page

About this book

This document is a set of connected examples (recipes, if you like) for data engineering using Atlas Data Federation from MongoDB. *All examples are in JavaScript and can be run from mongosh, the MongoDB shell, but it should be relatively easy for any programmer to translate them to Python, Java, or C#.*

Sometimes, you don't want a lot of story or explanation, just a code example. So I've tried to keep the word count low in this relative to the examples. However, it's still worth reading the explanations.

These examples deliberately use non trivial data set sizes so you can understand performance at real-world scale.

In this book, commands and their output are rendered as text, not images, so you can copy commands where appropriate. Data you enter is in white where output is typically shown in green with some highlighting. Do not simply copy and paste the entire contents of a box as this will include prompts and output.

Common ingredients

What is Atlas Data Federation?

Atlas Data Federation (ADF) is a way you can combine multiple MongoDB clusters, Amazon S3 buckets, Azure BLOB stores, and HTTP data sources into a single, virtual MongoDB database instance. ADF allows you to query non-MongoDB sources — such as CSV, JSON, or Parquet files — as though they were MongoDB databases and easily transfer data between MongoDB collections and files in S3 or Azure using aggregation and query commands.

You can use it to export data in multiple formats to feed systems that expect data delivery via S3. You can use it to import data from files in S3 or Azure or even directly from websites, and you can use it to create a tiered storage system so older data can be kept in much cheaper cloud object storage but still be accessed in your application. It can even be used to optimize data for analytic tasks by converting it to analytics optimized file formats.

What is Amazon S3?

Amazon S3 storage is a cloud object store and presents as a simplistic filesystem-like service. Unlike a normal filesystem, it works at file level, not block level. You can read all or part of a file but only replace entire files. It's accessed via an HTTP API, and so it has a much higher read latency than a typical local or network filesystem. Essentially, you can GET, POST, PUT, and DELETE files in a notional hierarchy, as well as listing what exists at a given point in the hierarchy.

The advantage of S3 is that compared to a network filesystem or a database, it is far cheaper. Amazon's virtual disk offering (Elastic Block Storage — EBS) costs $0.08/GB/month, and if used for an Atlas cluster, you need multiple copies for availability. Basic S3 storage has a high uptime and is only $0.023/GB/month. This means for larger data sets that are less frequently accessed, it can be considerably cheaper. This is a balancing act as unlike EBS, Amazon charges a small amount per read and write operation on the data — approximately $0.40 for every one million reads.

At the time of publication, Atlas Data Federation also supports Microsoft's Azure Blob storage which is a direct equivalent to Amazon S3 storage in the Azure environment.

Can I use Azure Blob Storage?

At the time of publication, MongoDB has announced Azure Blob storage as an alternative to AWS S3 for data federation. This is in private preview — with the exception of the specific commands to authorize Azure storage, these instructions should be almost identical when using Azure rather than AWS S3.

What is the MongoDB Shell?

The MongoDB Shell, mongosh, is a JavaScript command line tool based on Node.js that runs locally on your desktop or laptop. Code you run there is not running on the database server; it's running locally. Like any programming language you can connect to MongoDB (or other databases) and make API calls to read and write to them, we are using it here as it's the simplest universal coding environment.

Mongosh defaults to connection to a MongoDB cluster when you run it to save you explicitly opening one as you would in another environment. You can alternatively run these recipes from any other programming language and environment that has a MongoDB driver — for example, PHP, Java, C#, Node.js, or Python. They would need only small syntactic changes to suit your language.

JavaScript isn't everyone's favorite language but it's easy to read and most people have used the MongoDB shell at some point, so it is the default for these examples.

How does Data Federation work?

Data Federation works by MongoDB maintaining a fleet of query processing engines (with strong data isolation) running in a number of cloud provider regions. These present a subset of the MongoDB protocol API, which is the API calls underpinning operations like find() and aggregate(). These engines can forward operations to Atlas clusters or fetch data from S3, apply Parquet optimisations (including column projections), then convert into BSON (MongoDB's typed data format) before streaming it through a MongoDB query engine.

Each file read is processed in parallel, with a map-reduce algorithm used to combine the results behind the scenes. This brings you the power, scale, and resilience of map-reduce processing without the downsides of trying to write map and reduce functions to implement it.

Data Federation does not support direct write methods like insert or update even against MongoDB data sources as they would be inefficient. The only way to write out data is using $out and $merge in aggregation. It allows you to configure granular security and limit what users can read and write the same way you can for a MongoDB database, including limiting access to specific folders in cloud storage.

How does Data Federation charge?

Data federation charges per 1TB of data accessed when performing an operation regardless of whether it's read from HTTP, S3, Azure Blob, or Atlas. It has a minimum read of 10MB per operation. The price at the time of writing is $5 per TB accessed.

“Accessed” means any data that has to be read into the query engine to satisfy a query

Additionally, Data Federation charges for Data Returned and Transferred:

- The number of bytes transferred between Data Federation query nodes while executing a query.
- The number of bytes written by Data Federation during $out or $merge operations.
- The number of bytes returned to the user as query results.

ADF pushes as much of your query as possible down to any underlying database cluster. Efficiently projecting columns from columnar formats like Parquet, as well as using data partitioning and filename matching, minimizes the quantity of data read to keep costs as low as possible.

In addition to any ADF pricing, you will pay cloud provider data egress charges from your bucket to the location of the MongoDB processing fleet if your S3 storage is not in one of the regions where MongoDB hosts query processing engines.

The recipes

Required equipment

To reproduce these recipes, you will need:

- A MongoDB Atlas cluster.**
- AtlasAdmin permissions for the AWS cluster.
- An AWS account that you have administrative permissions for.
- The MongoDB shell (mongosh) installed.
- The AWS command line tool awscli installed and configured.

You can use ADF with an M0 free-tier Atlas cluster but these examples work with a lot of data, and the shared tier clusters M0 through M5 are heavily throttled for disk size and query performance. You need at least an M10 if you are using the data I use in these examples. You also need to provision at least 128GB of storage. **It is strongly recommended to disable auto-scaling on your Atlas cluster when experimenting as large-scale, one-off data movements will be detected as a need to upscale the cluster.

Enabling Data Federation and attaching it to AWS S3

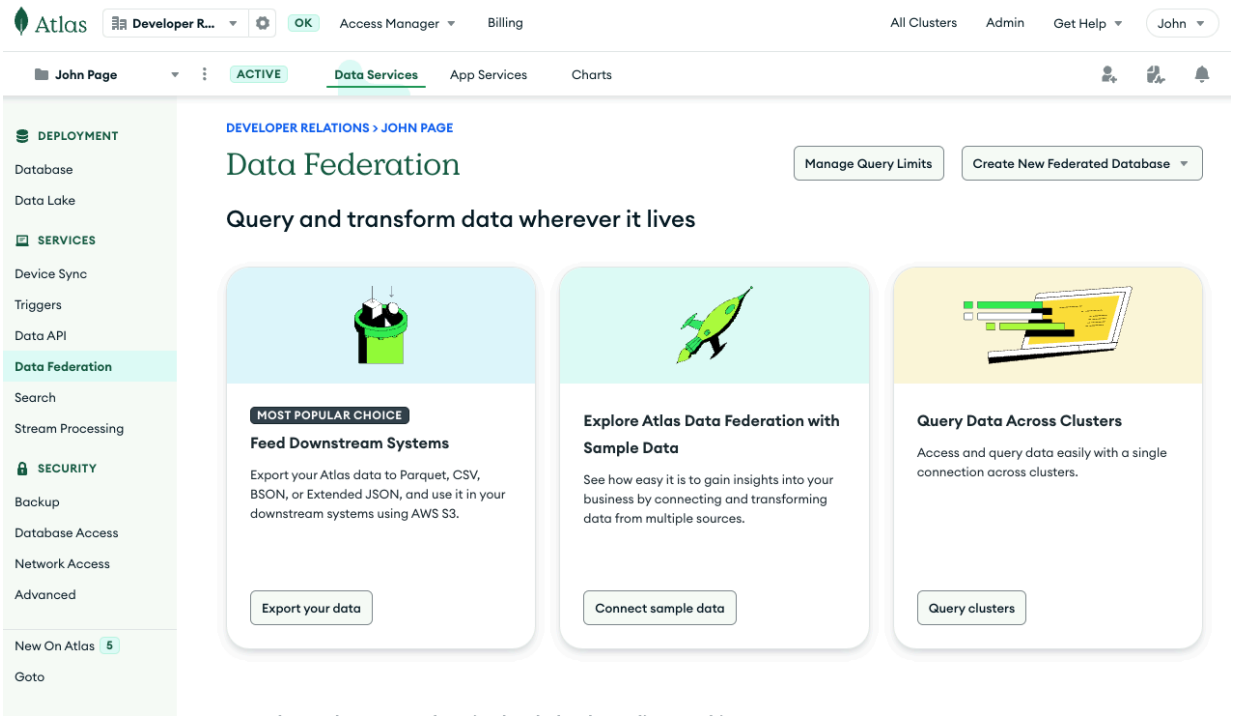
This section is brief and the only one where we use the Atlas GUI. The reason for using the GUI here is that we need to create an S3 bucket and grant the MongoDB AWS account Data Federation runs as permissions to read and write it. There is a nice step-by-step tutorial in the GUI to walk you through this if you are not familiar with configuring AWS permissions and roles.

1. First, ensure you have created an S3 bucket in the region where you want to store your data and have installed and configured the AWS CLI tools as you will need them to configure security. You can verify you have done both of these things by using the AWS command line tools. I have created a bucket called "jpage-datalake" in the "eu-west-1" region of AWS and can verify that as follows.

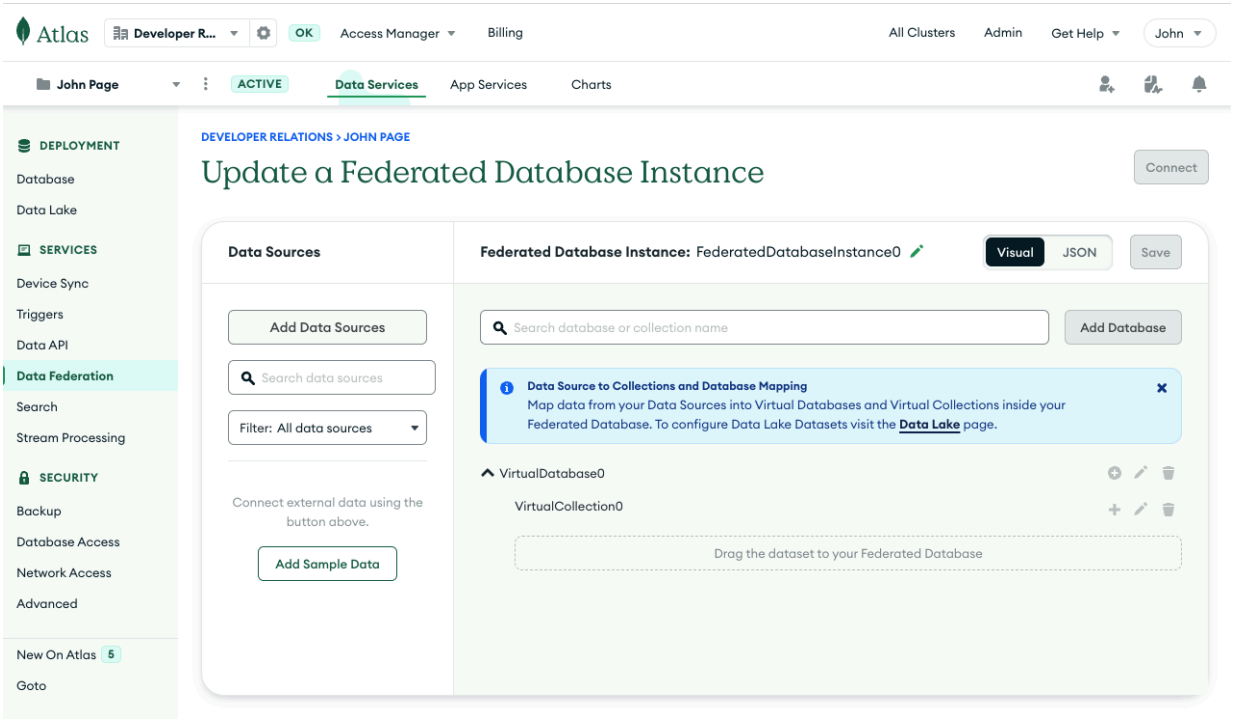
```
laptop> aws s3api get-bucket-location --bucket  jpage-datalake
{
    "LocationConstraint": "eu-west-1"
}
```

2. Ensure you have a dedicated Atlas Cluster — ideally, in the same region with 128GB or more of storage and both Tier and Storage scaling disabled. I have an M10 in eu-west-1 (Ireland) called Cluster0 which costs $0.16 per hour.

3. From the Atlas homepage, click "Data Federation" on the left side menu bar.

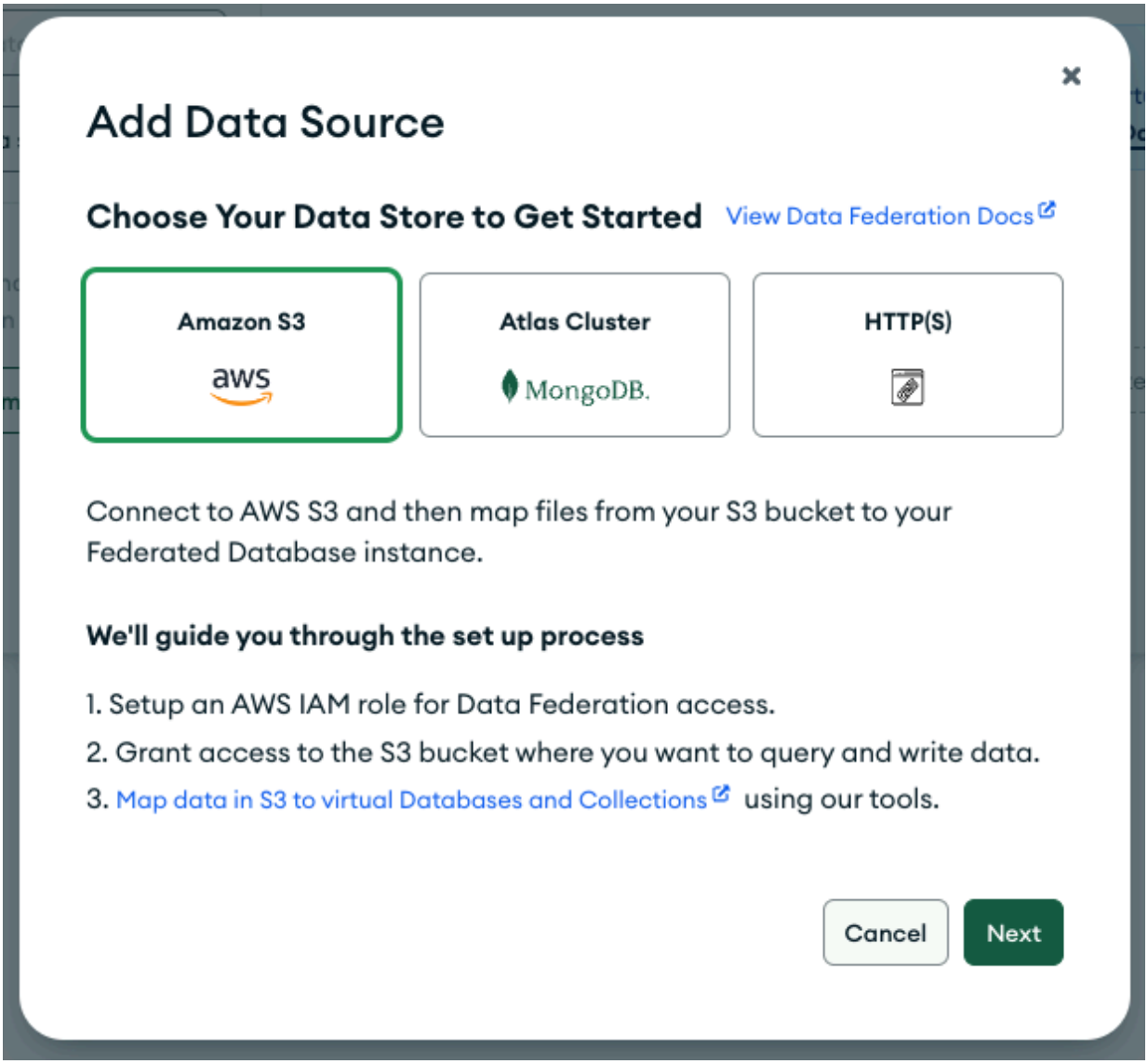


4. Select "Create New Federated Database - Set up Manually."

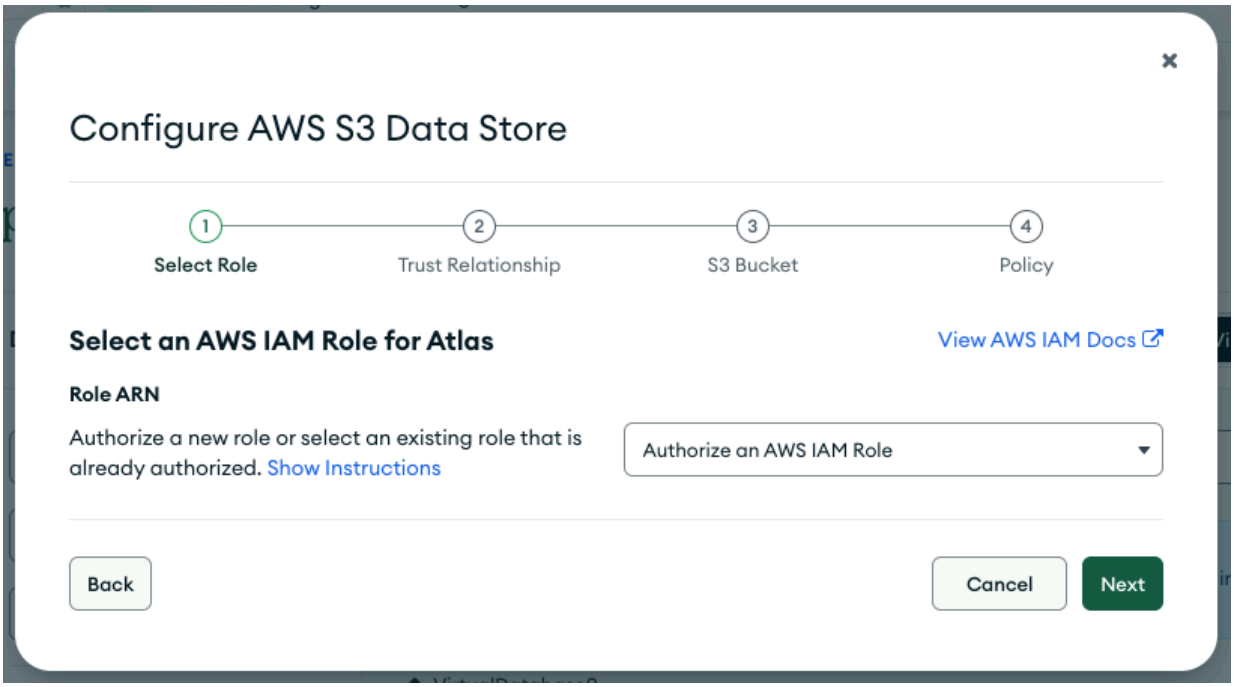


5. On the next screen, click "Add Data Source." Then, follow the steps in the wizard to attach your S3 bucket.

1. Select Amazon S3 and click “Next.”



2. If this is the first time you have done this, you will be taken through a set of instructions to authorize Atlas Data Federation's access to your S3 bucket (authorize an AWS IAM Role). Follow the steps in this wizard.


Configure AWS S3 Data Store
1
Select Role
2
Trust Relationship
3
S3 Bucket
4
Policy
Select an AWS IAM Role for Atlas
View AWS IAM Docs
Role ARN
Authorize a new role or select an existing role that is already authorized. Show Instructions
Authorize an AWS IAM Role
Back
Cancel
Next

Create New Role with the AWS CLI

1. Copy and save this JSON file as `role-trust-policy.json`. It describes the trust relationships that allows Atlas to assume your new AWS IAM role.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::536727724300:root"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "44577bd8-ed7d-47ec-91b9-d72b43946ccf"
        }
      }
    }
  ]
}
```

2. Enter a name for your new AWS IAM role. This name will auto-populate the command in the next step.

atlas-data-lake-role

3. Copy and enter the command below into the AWS CLI to create your new IAM role for Atlas.

```
aws iam create-role \
 --role-name atlas-data-lake-role \
 --assume-role-policy-document file://role-trust-policy.json
```

4. Enter the Role ARN

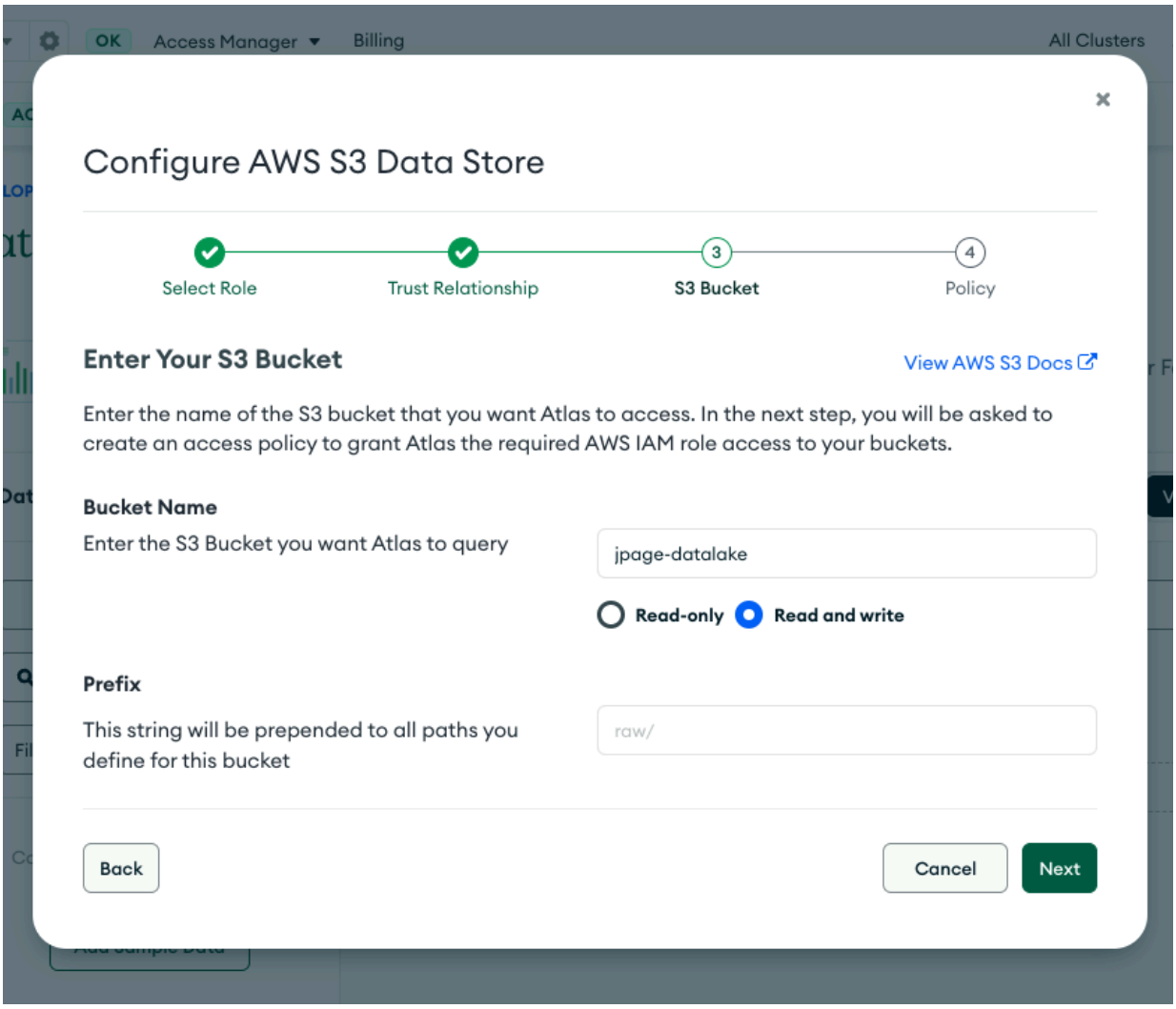
This is the role ARN for the AWS IAM role you just created or updated that Atlas will use to access your AWS environment.

arn:aws:iam::123456789012:role/roleName

```
laptop> cat >  role-trust-policy.json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Principal": {
        "AWS": "arn:aws:iam::536727724300:root"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "44577bd8-ed7d-47ec-91b9-d72b43946ccf"
        }
      }
    }
  ]
} <Control-D-To-Exit>


laptop> aws iam create-role \
  --role-name atlas-data-lake-role \
  --assume-role-policy-document file://role-trust-policy.json
{
    "Role": {
        "Path": "/",
        "RoleName": "atlas-data-lake-role",
        "RoleId": "AROA2QKJSCXBZ6N2F4ZNA",
        "Arn": "arn:aws:iam::722245653955:role/atlas-data-lake-role",
        "CreateDate": "2023-09-15T08:47:10+00:00",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "AWS": "arn:aws:iam::536727724300:root"
                    },
                    "Action": "sts:AssumeRole",
                    "Condition": {
                        "StringEquals": {
                            "sts:ExternalId":
"44577bd8-ed7d-47ec-91b9-d72b43946ccf"
                        }
                    }
                }
            ]
        }
    }
}
```

3. On this screen, enter your bucket name, select "Read and Write," and leave the prefix blank. My bucket is called “jpage-datalake” and I will use that through the rest of these examples. *As bucket names are globally unique, you will need to replace it with your own bucket name in all the recipes.* Leave the prefix blank at this stage.



You will then be prompted to configure specific access for ADF to the bucket in AWS.

Configure AWS S3 Data Store

Select Role — Trust Relationship — S3 Bucket — 4 Policy

Assign an access policy to your AWS IAM role

View AWS IAM Docs

1. Copy and save this JSON file as `adl-s3-policy.json`. It describes the role policy that allows Atlas to access your AWS resources.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:GetBucketLocation"
      ],
      "Resource": [
        "arn:aws:s3:::jpage-datalake",
        "arn:aws:s3:::jpage-datalake/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws:s3:::jpage-datalake",
        "arn:aws:s3:::jpage-datalake/*"
      ]
    }
  ]
}
```

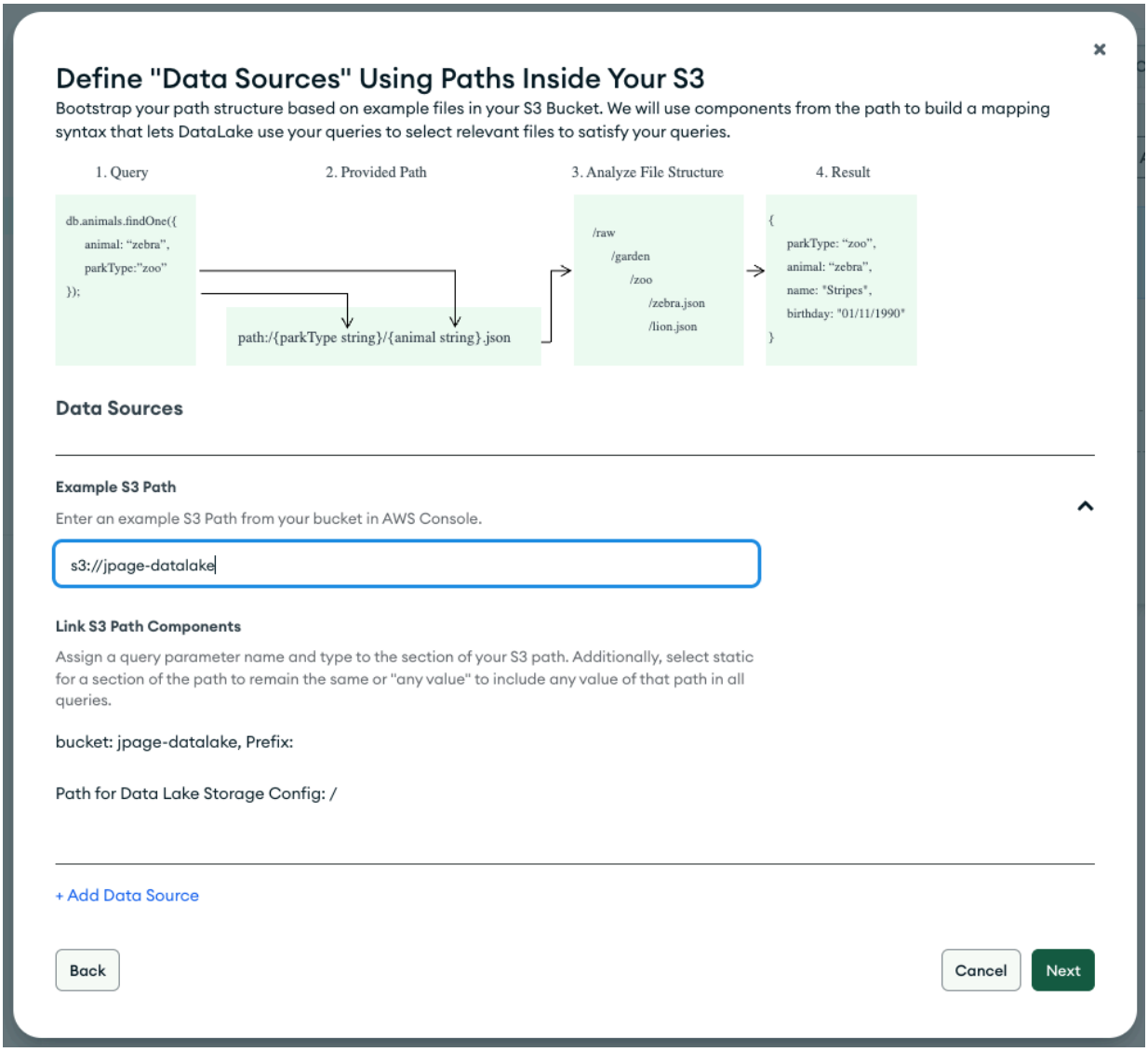
2. Copy and enter the command below into the AWS CLI to attach the role policy to your AWS IAM role.

```
aws iam put-role-policy \
  --role-name atlas-data-lake-role \
  --policy-name atlas-data-lake-role-policy \
  --policy-document file://adl-s3-policy.json
```

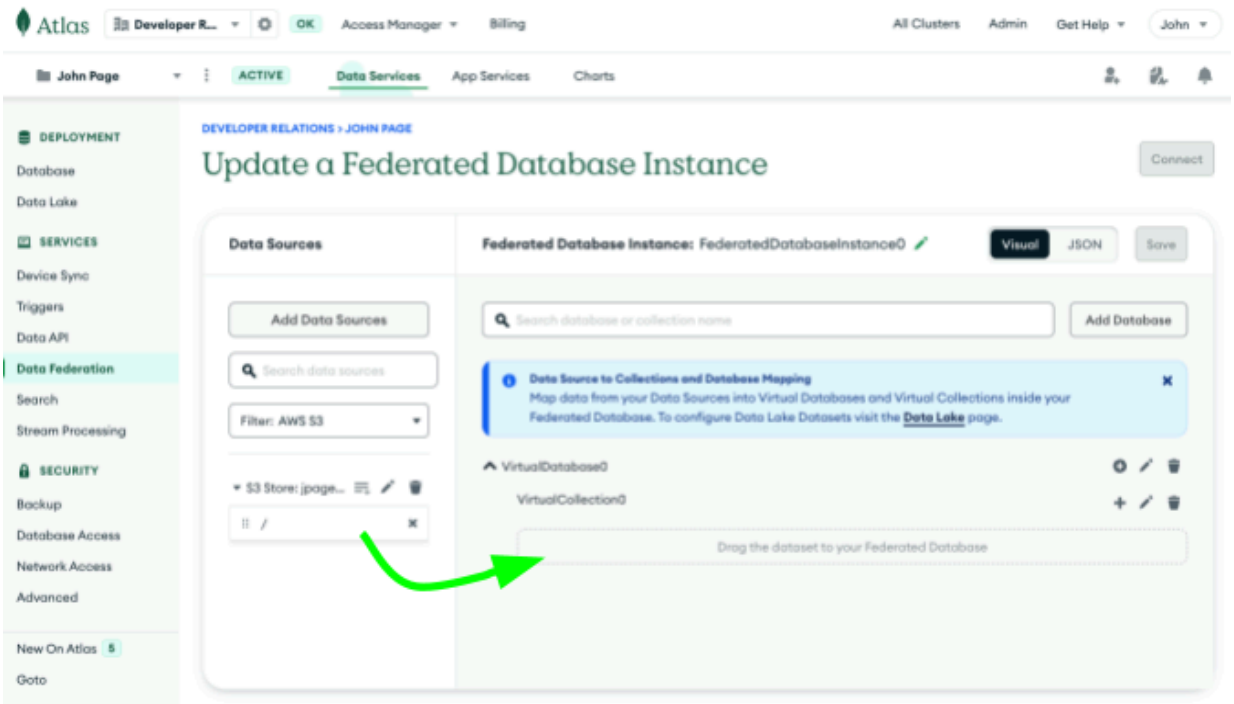
Back | Cancel | Next

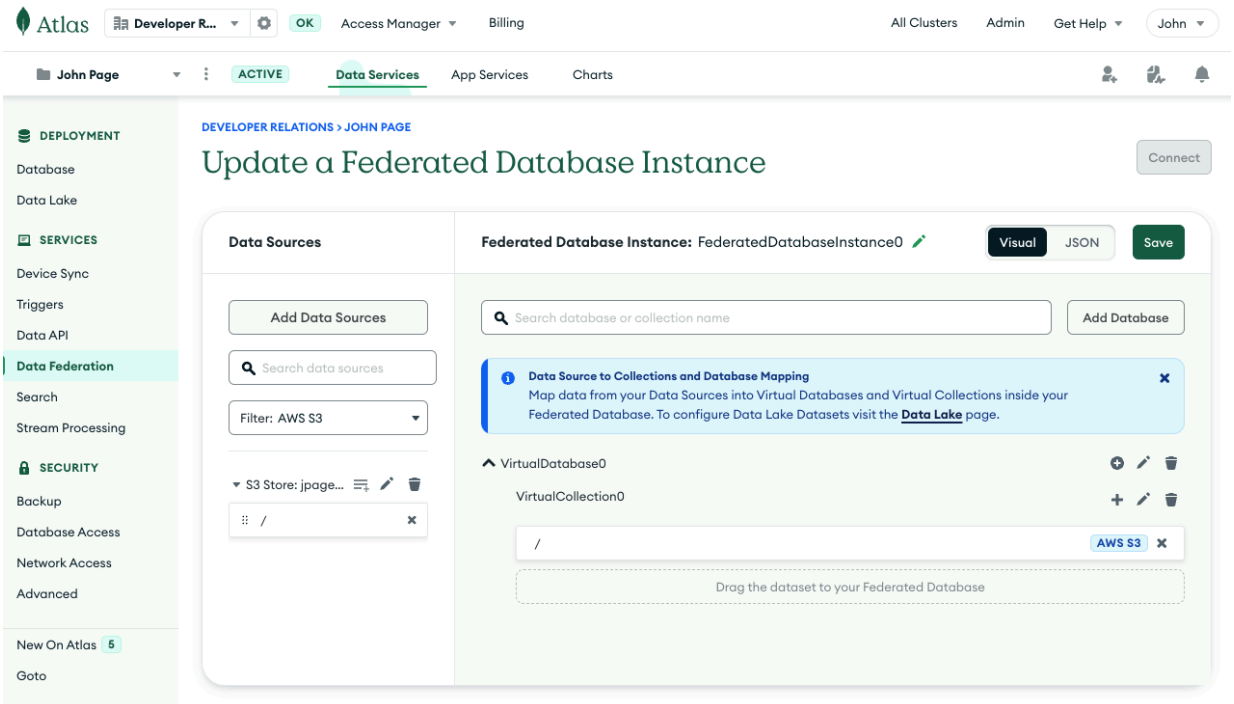
```
laptop> cat > adl-s3-policy.json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:GetBucketLocation"
      ],
      "Resource": [
        "arn:aws:s3:::jpage-datalake",
        "arn:aws:s3:::jpage-datalake/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws:s3:::jpage-datalake",
        "arn:aws:s3:::jpage-datalake/*"
      ]
    }
  ]
} <Control-D-to-Exit

laptop> aws iam put-role-policy \
   --role-name atlas-data-lake-role \
   --policy-name atlas-data-lake-role-policy \
   --policy-document file://adl-s3-policy.json
laptop>
```

4. After clicking "Next," enter "s3://<your bucket name>" in the example S3 path. We will configure this in more detail later.

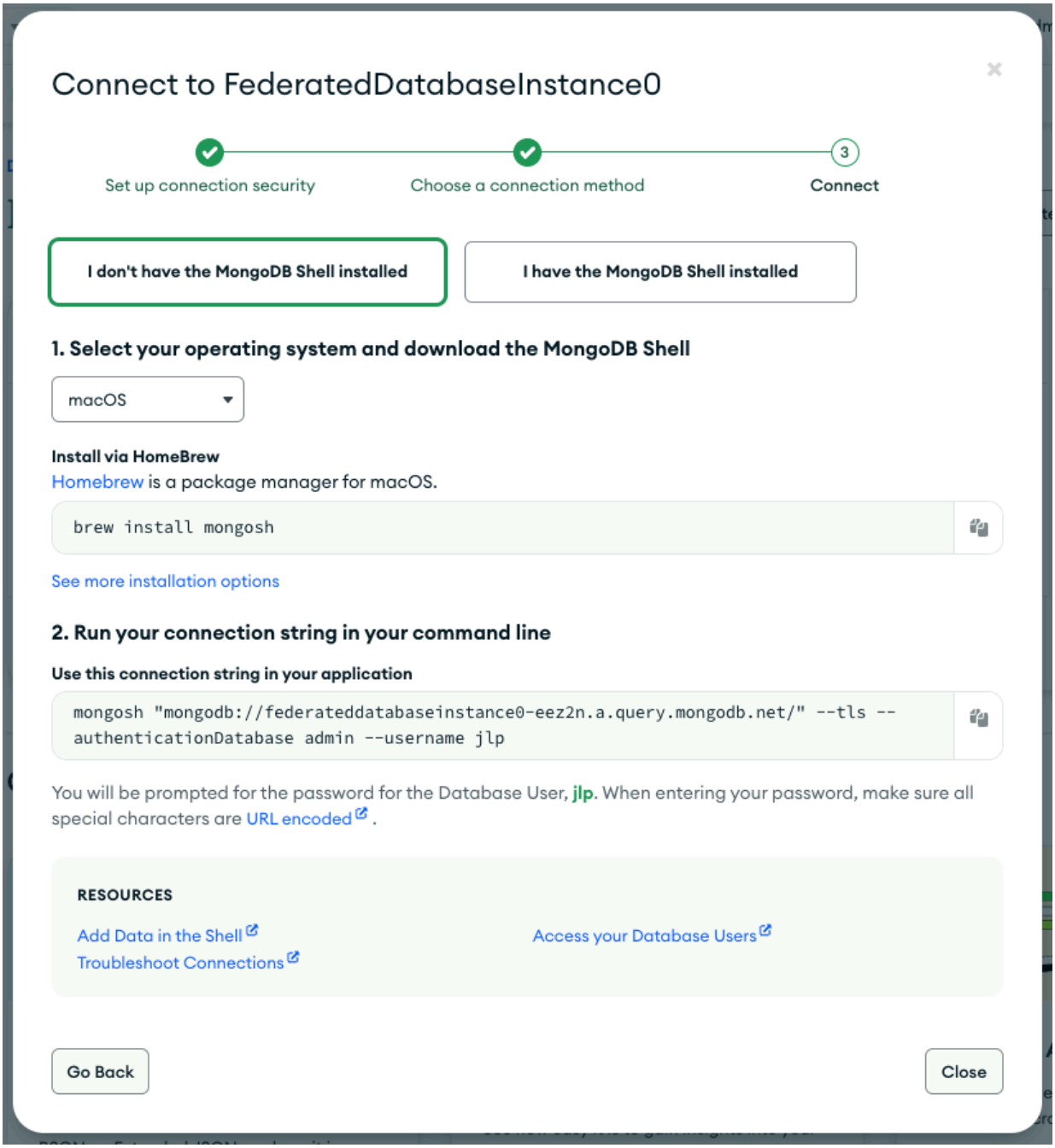


5. Finally, you need to drag the dataset to the place that says "Drag the dataset to your Federated Database." This will make the whole S3 bucket available as a collection called “VirtualCollection0” in a database called "VirtualDatabase0." We will rename those later. Once you do this, you can click “Save.”


Atlas
Developer R...
OK
Access Manager
Billing
All Clusters
Admin
Get Help
John
John Page
ACTIVE
Data Services
App Services
Charts
DEPLOYMENT
Database
Data Lake
SERVICES
Device Sync
Triggers
Data API
Data Federation
Search
Stream Processing
SECURITY
Backup
Database Access
Network Access
Advanced
New On Atlas 5
Goto
DEVELOPER RELATIONS > JOHN PAGE
Update a Federated Database Instance
Connect
Data Sources
Federated Database Instance: FederatedDatabaseInstance0
Visual
JSON
Save
Add Data Sources
Search data sources
Filter: AWS S3
S3 Store: jpage...
/
Search database or collection name
Add Database
Data Source to Collections and Database Mapping
Map data from your Data Sources into Virtual Databases and Virtual Collections inside your Federated Database. To configure Data Lake Datasets visit the Data Lake page.
VirtualDatabase0
VirtualCollection0
Drag the dataset to your Federated Database


Atlas
Developer R...
OK
Access Manager
Billing
All Clusters
Admin
Get Help
John
John Page
ACTIVE
Data Services
App Services
Charts
DEPLOYMENT
Database
Data Lake
SERVICES
Device Sync
Triggers
Data API
Data Federation
Search
Stream Processing
SECURITY
Backup
Database Access
Network Access
Advanced
New On Atlas 5
Goto
DEVELOPER RELATIONS > JOHN PAGE
Update a Federated Database Instance
Connect
Data Sources
Federated Database Instance: FederatedDatabaseInstance0
Visual
JSON
Save
Add Data Sources
Search data sources
Filter: AWS S3
S3 Store: jpage...
/
Search database or collection name
Add Database
Data Source to Collections and Database Mapping
Map data from your Data Sources into Virtual Databases and Virtual Collections inside your Federated Database. To configure Data Lake Datasets visit the Data Lake page.
VirtualDatabase0
VirtualCollection0
/
AWS S3
Drag the dataset to your Federated Database

6. Finally, click "Save," "Connect," and then “Connect with the MongoDB shell." Follow the instructions to get a shell connected to your federated instance.



When you are successful, it should look like this:

```
laptop>  mongosh
"mongodb://federateddatabaseinstance0-eez2n.a.query.mongodb.net/" --tls
--authenticationDatabase admin --username jlp
Enter password: **********
Current Mongosh Log ID:    65041f77a1eaa58869db3b97
Connecting to:
mongodb://<credentials>@federateddatabaseinstance0-eez2n.a.query.mongodb.
net/?directConnection=true&tls=true&authSource=admin&appName=mongosh+1.10
.1
Using MongoDB:        7.0.0
Using Mongosh:        1.10.1
For mongosh info see: https://docs.mongodb.com/mongodb-shell/
Warning: Found ~/.mongorc.js, but not ~/.mongoshrc.js. ~/.mongorc.js will
not be loaded.
  You may want to copy or rename ~/.mongorc.js to ~/.mongoshrc.js.
AtlasDataFederation test>
```

Modifying ADF configuration programmatically

We will not be using the ADF GUI after this point. We can view and modify the configuration and view logs' information entirely from the shell or any other coding environment. If you return at any point to the GUI, you will see all the updates we have made.

We retrieve the federation config with the **storageGetConfig** admin command. This needs a relatively high level of permissions so for simplicity, at this stage you should have an Atlas account with atlasAdmin permissions.

To make later examples easier to read, we can change the prompt in our shell to just **ADF>** using the following:

```
AtlasDataFederation test> prompt=function(){return "ADF> "}
[Function: prompt]
ADF>
```

```
const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand( { "storageGetConfig" : 1 } ).storage
console.log(fedconf)
```

```
ADF> console.log(fedconf)
{
  stores: [
    {
      name: 'jpage-datalake',
      provider: 's3',
      region: 'eu-west-1',
      delimiter: '/',
      bucket: 'jpage-datalake'
    }
  ],
  databases: [
    {
      name: 'VirtualDatabase0',
      collections: [
        {
          name: 'VirtualCollection0',
          dataSources: [ { storeName: 'jpage-datalake', path: '/' } ]
        }
      ]
    }
  ]
}
```

We can modify the federation configuration object using code in the shell to rename our S3 datastore, virtual database, and virtual collection from their default names. After changing the fedconf object we then write it back using the storageSetConfig command.

```
fedconf.stores[0].name = 's3bucket'
fedconf.databases[0].name='nhs' // We will demo using UK medical data
fedconf.databases[0].collections[0].name='rawdata'
```

```
fedconf.databases[0].collections[0].dataSources[0].storeName='s3bucket'

console.log(fedconf)
adminDB.runCommand( { "storageSetConfig" : fedconf } )
```

```
ADF> console.log(fedconf)
{
  stores: [
    {
      name: 's3bucket',
      provider: 's3',
      region: 'eu-west-1',
      delimiter: '/',
      bucket: 'jpage-datalake'
    }
  ],
  databases: [
    {
      name: 'nhs',
      collections: [
        {
          name: 'rawdata',
          dataSources: [ { storeName: 's3bucket', path: '/' } ]
        }
      ]
    }
  ]
}
ADF> adminDB.runCommand( { "storageSetConfig" : fedconf } )
{ ok: 1 }
```

Now, we can use "show databases," "use nhs," and "show tables" to see what we have configured. There is no data in there yet. We can do that in the next step.

```
ADF> show dbs
nhs  0 B
ADF> use nhs
switched to db nhs
ADF> show tables
rawdata
ADF>
```

Copying data into an S3 bucket from a public HTTP source

Note - This step takes 3.5 hours to complete.

We need to bootstrap our examples with some data. To make this meaningful, I'm going to put a non-trivial quantity of data into my S3 bucket to get started. The UK government publishes the details of every single prescription written by a doctor in the UK every month and we have 12 years of this data, so to illustrate, I am going to load that data in.

We could use the MongoDB shell with a couple of node modules to download the data to our own computer using the data.gov.uk APIs and then push it to S3 using the AWS APIs or command line tools, but that would mean downloading and then uploading hundreds of gigabytes of data via my internet connection, which could take a very long time.

We can take advantage of the fact that Data Federation can read data from a publicly visible HTTPS source to copy it directly into S3. To do this, we will retrieve the list of files available through the open data API using a call to the API from our local machine. Then, for each file, configure it as an HTTP data source in Federation.

Then, we will use an aggregation pipeline to read the contents from the API and write them directly to our S3 bucket. We will discuss this more in a later step. MongoDB's Data Federation servers will be much quicker at downloading and writing the data than routing data via your internet connection. We can also take the opportunity to compress the CSV files as we write them, saving us both S3 storage and read costs.

If you are doing this with your own private data, you would upload it to S3 directly using AWS tools as Data Federation can only see public HTTP servers.

The code below can be pasted into the mongo shell, or saved as a `getnhsdata.js` file and run with `load('getnhsdata.js')` in the shell. It is probably easier to save it as a js file.

getnhsdata.js

```
/* This scripts fetches a list of urls from the web using an API
then sets each up as a virtual datastore - it does not download much data.
To run this you need to have installed axios (So have npm installed too)
npm install axios */


bucket = 'jpage-datalake'
region = 'eu-west-1'


function getHttpStore() {
  // If we don't have http configured as a data store then add it
  const adminDB = db.getSiblingDB('admin')
  const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage
  const stores = fedconf.stores
  let httpStore = stores.find(ds => ds.provider === 'http')
  if (!httpStore) {
    httpStore = { name: 'https', provider: 'http' }
    stores.push(httpStore)
    adminDB.runCommand({ storageSetConfig: fedconf })
  }
  return httpStore
}

function attachHttpAsCollection(fedconf, storename,
  dbname, url, s3path) {
  // Add our HTTP page as a temporary HTTP data source
  // Add each file as a different collection to allow parallelism
  url = url.replace('http:', 'https:')


  // Add data.gov.uk as a database if it's not there
  let databases = fedconf.databases
  let virtualdb = databases.find(db => db.name === dbname)
  if (!virtualdb) {
    virtualdb = { name: dbname, collections: [] }
    databases.push(virtualdb)
```

```
    console.log(`Adding virtual DB ${virtualdb.name}`)
  }
  let collections = virtualdb.collections
  // Add this as a virtual collection if it's not already there
  s3path = s3path.replace(/\//g, '_')
  let virtualCollection = collections.find(co => co.name === s3path)
  if (!virtualCollection) {
    virtualCollection = { name: s3path }
    virtualCollection.dataSources = [{ storeName: storename, urls: [url] }]
    collections.push(virtualCollection)
  }
}


async function fetchData() {
  const axios = require('axios')
  const httpStore = getHttpStore()
  console.info('Fetching the catalog of files')
  const catalogueURL =
'https://data.gov.uk/api/action/package_search?fq=name:prescribing-by-gp-practic
e-presentation-level'


  const csvList = []
  try {
    const catalog = await axios.get(catalogueURL)
    const adminDB = db.getSiblingDB('admin')
    const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage
    for (const resource of catalog?.data?.result?.results[0]?.resources) {
      if (resource.name !== null && resource.mimetype === 'text/csv'
        && resource.name.includes('prescribing')) {
        const [all, month, year, desc] = resource.name.match(/^(\w*) (\d{4})
(.*)/)
        const s3pathname = `rawdata/${desc.replace(/ /g,
'_')}/${year}/${month}`.toLowerCase()


        // console.info(s3pathname)
        attachHttpAsCollection(fedconf, httpStore.name, 'datagovuk',
          resource.url, s3pathname)
        csvList.push(s3pathname)
      }
    }
    adminDB.runCommand({ storageSetConfig: fedconf })
    const httpvirtualdb = db.getSiblingDB('datagovuk')
    for (const csvfile of csvList) {
      // Write data to S3 using $out
      console.log(`Copying ${csvfile} from HTTP Source to S3`)
      const writeCSVandCompress = {
        $out: {
```

```
          s3: {
            bucket,
            region,
            filename: `${csvfile}`,
            format: { name: 'csv.gz', maxFileSize: '2GB' },
            errorMode: 'stop'
          }
        }
      }

      const virtualcollectionname = csvfile.replace(/\//g, '_')
      httpvirtualdb[virtualcollectionname].aggregate([writeCSVandCompress])
    }
  } catch (e) {
    print(`Error Caught: ${e}`)
  }
}


fetchData()
```

Once the HTTP sources are configured, reading from them and writing out to S3 can take up to 12 hours to complete. Please ensure your client has a very stable connection to the database server in that time. *One option that I have used here is to start a very small ec2 instance and use that to run mongosh. The data is still not routed via the ec2-instance but the network will not drop the way home internet might. To do this, I have to install npm and then Axios as my script requires them to fetch the list of URLs to copy from a data.gov API.*

```
[ec2-user@ip-172-31-13-238 ~]  sudo yum install
https://rpm.nodesource.com/pub_16.x/nodistro/repo/nodesource-release-nodi
stro-1.noarch.rpm -y
[ec2-user@ip-172-31-13-238 ~]  sudo yum install nodejs -y
--setopt=nodesource-nodejs.module_hotfixes=1
[ec2-user@ip-172-31-13-238 ~] npm install axios
[ec2-user@ip-172-31-13-238 ~]  curl -OL
https://downloads.mongodb.com/compass/mongosh-1.10.6-linux-x64.tgz
[ec2-user@ip-172-31-13-238 ~] tar xvzf mongosh-1.10.6-linux-x64.tgz
```

```
[ec2-user@ip-172-31-13-238 ~] cd mongosh-1.10.6-linux-x64/bin
[ec2-user@ip-172-31-13-238 bin]$ ./mongosh
"mongodb://federateddatabaseinstance0-eez2n.a.query.mongodb.net/" --tls
--authenticationDatabase admin --username jlp
Enter password: **********
Current Mongosh Log ID: 6504327df40a1f85bce25c85
Connecting to:
mongodb://<credentials>@federateddatabaseinstance0-eez2n.a.query.mongodb.
net/?directConnection=true&tls=true&authSource=admin&appName=mongosh+1.10
.6
Using MongoDB:          7.0.0
Using Mongosh:          1.10.6
mongosh 2.0.1 is available for download:
https://www.mongodb.com/try/download/shell
For mongosh info see: https://docs.mongodb.com/mongodb-shell/
To help improve our products, anonymous usage data is collected and sent
to MongoDB periodically (https://www.mongodb.com/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.
AtlasDataFederation test>
console.time();load("getnhsdata.js");console.timeEnd()
Fetching the catalog of files
Adding virtual DB datagovuk
Copying rawdata/practice_prescribing_data/2010/august from HTTP Source to
S3
```

Once we have done this, we can remove the HTTP data sources from the config to keep it small and readable. We are not going to want to continually access the data from the original HTTP API anyway. *Functions, such as dropVirtualDB defined here, are intended to be reused in your own code. You can store them in a library or in your mongosh.rc file so they are always available.*

```
function dropVirtualDB(dbname) {
     const adminDB = db.getSiblingDB('admin')
```

```
     const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage
     let virtualdb = fedconf.databases.findIndex(db => db.name ===
dbname)
     if (virtualdb != -1) {
        fedconf.databases.splice(virtualdb, 1);
     }
     console.log(fedconf)
     adminDB.runCommand({ storageSetConfig: fedconf })
 }

dropVirtualDB("datagovuk")
```

```
ADF> dropVirtualDB("datagovuk")
{
  stores: [
    {
      name: 's3bucket',
      provider: 's3',
      region: 'eu-west-1',
      delimiter: '/',
      bucket: 'jpage-datalake'
    },
    { name: 'https', provider: 'http' }
  ],
  databases: [
    {
      name: 'nhs',
      collections: [
        {
          name: 'rawdata',
          dataSources: [ { storeName: 's3bucket', path: '/' } ]
        }
      ]
    }
  ]
}
```

Accessing data from files in S3

When we have data files in CSV, JSON, or other supported formats in S3, we can access them as though they were a virtual collection. Here we will create a single collection from the data we copied to S3 in the previous recipe. This is the simplest way to access the data but further manipulation to improve the file structure and data quality can significantly add value and reduce access times and costs, as we will see in later recipes .

To set up the raw CSV files as a single large virtual collection that we can then read, we add the entire directory with a wildcard like so.

```
const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand( { "storageGetConfig" : 1 } ).storage
const dbname = 'nhs'
const collectionname = 'rawcsvprescribingdata'
const dataSources =[{ storeName: 's3bucket', path: '/rawdata/*' }]
const virtualdb = fedconf.databases.find( db => db.name==dbname)
if(virtualdb) {
     //Add or replace in array
     collection = virtualdb.collections.find(c => c.name==collectionname)
     if(collection) {
        collection.dataSources = dataSources
     } else {
        virtualdb.collections.push({name:collectionname,dataSources})
     }
}
console.log(fedconf)
adminDB.runCommand( { "storageSetConfig" : fedconf } )
```

```
ADF> console.log(fedconf)
{
  stores: [
    {
      name: 's3bucket',
      provider: 's3',
      region: 'eu-west-1',
      delimiter: '/',
      bucket: 'jpage-datalake'
    },
    { name: 'https', provider: 'http' }
  ],
  databases: [
    {
      name: 'nhs',
      collections: [
        {
          name: 'rawdata',
          dataSources: [ { storeName: 's3bucket', path: '/' } ]
        },
        {
          name: 'rawcsvprescribingdata',
          dataSources: [ { storeName: 's3bucket', path: '/rawdata/*' } ]
        }
      ]
    }
  ]
}
ADF> adminDB.runCommand( { "storageSetConfig" : fedconf } )
{ ok: 1 }
```

We can now use show tables (or show collections) to see our newly configured source and findOne to see a record from it. It's not very nice looking yet, though.

```
ADF> show dbs
nhs  0 B
ADF> use nhs
already on db nhs
ADF> show tables
rawcsvprescribingdata
rawdata
ADF> db.rawcsvprescribingdata.findOne()
{
  '                ': '                  ',
  ' SHA': 'Q44',
  'ACT COST   ': '00000002.12',
  'BNF CODE': '0401010Z0AAAAAA',
  'BNF NAME                                ': 'Zopiclone_Tab 7.5mg
',
  'ITEMS  ': '0000006',
  'NIC        ': '00000001.56',
  PCT: 'RTV',
  PERIOD: '201804',
  PRACTICE: 'Y04937',
  QUANTITY: '0000063'
}
ADF>
```

Clean up a raw CSV data file

The data we downloaded and made available from the NHS, in common with many raw data files — especially those in CSV — is very ugly. As we can see above, the data did not have quote delimiters round fields, had an extra comma at the end, and had fixed width fields padded with spaces and spaces in some column names — none of which make it easy to work with.

In addition, Data Federation, by default, reads every column as a string as it cannot know the data type, and inferring it may lead to issues if the values are normally numeric but then contain missing data or a string like "null." The first thing we need to do with our raw data, therefore, is to apply a view on top of it that filters and cleans the data as it is read.

Atlas Data Federation allows you to create multiple views on a given collection. A view is defined as a MongoDB aggregation pipeline but stored in the config file as a string. Here we create a subtable pipeline programmatically to transform our raw data and then configure it as a view.

In this view, we will:

- Remove spaces from field names.
- Remove trailing spaces in text fields.
- Remove the extra field caused by a trailing comma.
- Convert numeric fields to numeric data types.
- Lowercase field names (purely for aesthetic reasons).

createcleanview.js

```
// Trim all field names both ends and all values at the right
// Also lowercase fieldnames
// We don't know what fields are there so we will convert the Object to an
// array of  key value pairs then back to an object

const recordAsArrayOfFields = { $objectToArray: '$$ROOT' }

// Single fields are  {k: "Key Name", v: "Value" } after $arrayToObject

const despacekey = { $replaceAll: { input: '$$field.k',
                                    find: ' ', replacement: '' } }

const cleanSingleField = { k: { $toLower: despacekey },
                           v: { $trim: { input: '$$field.v' } } }

const spacesRemoved = { $map: { input: recordAsArrayOfFields,
                                as: 'field', in: cleanSingleField } }

// We had a comma at the end of our CSV so a column with a name that's
// just spaces and no data we can remove that from the array of fields

const removeEmptyFields = { $filter: { input: spacesRemoved, as: 'field',
                                       cond: { $ne: ['$$field.k', ''] } } }

const asObject = { $arrayToObject: removeEmptyFields }
const toRootObj = { $replaceRoot: { newRoot: asObject } }

// Convert some values to numbers - This could be done in the mapping
// above but we can make it a separate stage in the transform
```

```
const numericfields = ['actcost', 'items', 'nic', 'quantity']
const numberCasts = {}
for (const n of numericfields) {
  // Making them doubles but we have Int32,Int64 and Decimal128 available too
  numberCasts[n] = { $convert: { input: `$${n}`, to: 'double',
                                 onError: 0, onNull: 0 } }
}

const convertAllNumbers = { $set: numberCasts }
const viewPipeline = [toRootObj, convertAllNumbers]


// Add our view to the nhs database

const dbname = 'nhs'
const basecollection = 'rawcsvprescribingdata'
const viewname = 'cleanrawdata'
const viewPipelineAsJsonString = JSON.stringify(viewPipeline)

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage
const virtualdb = fedconf.databases.find(db => db.name === dbname)

if (virtualdb) {
  // Add or replace view in array
  if (!virtualdb.views) { virtualdb.views = [] }

  const view = virtualdb.views.find(vw => vw.name === viewname)
  if (view) {
    view.source = basecollection
    view.pipeline = viewPipelineAsJsonString
  } else {
    virtualdb.views.push({ name: viewname, source: basecollection,
                           pipeline: viewPipelineAsJsonString })
  }
}
console.log(fedconf)

adminDB.runCommand({ storageSetConfig: fedconf })
```

Running this creates a new view called cleanrawdata, visible when we use "show tables." Querying or aggregating against that view applies the transformations we need.

```
ADF>  console.log(fedconf)
{
  stores: [
    {
      name: 's3bucket',
      provider: 's3',
      region: 'eu-west-1',
      delimiter: '/',
      bucket: 'jpage-datalake'
    },
    { name: 'https', provider: 'http' }
  ],
  databases: [
    {
      name: 'nhs',
      collections: [
        {
          name: 'rawdata',
          dataSources: [ { storeName: 's3bucket', path: '/' } ]
        },
        {
          name: 'rawcsvprescribingdata',
          dataSources: [ { storeName: 's3bucket', path: '/rawdata/*' } ]
        }
      ],
      views: [
        {
          name: 'cleanrawdata',
          source: 'rawcsvprescribingdata',
          pipeline:
'[{"$replaceRoot":{"newRoot":{"$arrayToObject":{"$filter":{"input":{"$map
":{"input":{"$objectToArray":"$$ROOT"},"as":"field","in":{"k":{"$toLower"
:{"$replaceAll":{"input":"$$field.k","find":"
","replacement":""}}},"v":{"$trim":{"input":"$$field.v"}}}}},"as":"field"
,"cond":{"$ne":["$$field.k",""]}}}}}},{"$set":{"actcost":{"$convert":{"in
put":"$actcost","to":"double","onError":0,"onNull":0}},"items":{"$convert
":{"input":"$items","to":"double","onError":0,"onNull":0}},"nic":{"$conve
rt":{"input":"$nic","to":"double","onError":0,"onNull":0}},"quantity":{"$
convert":{"input":"$quantity","to":"double","onError":0,"onNull":0}}}}]'
        }
      ]
    }
  ]
}
ADF> adminDB.runCommand({ storageSetConfig: fedconf })
```

```
{ ok: 1 }
ADF> use nhs
switched to db nhs
ADF> db.cleanrawdata.findOne()
{
  sha: 'Q60',
  actcost: 7.45,
  bnfcode: '1001010P0AAAHAH',
  bnfname: 'Naproxen_Tab E/C 250mg',
  items: 1,
  nic: 8.06,
  pct: '05G',
  period: '201504',
  practice: 'M83015',
  quantity: 112
}
```

Adding a missing header line to CSV

Atlas Data Federation relies on CSV files having a first line with the name of the fields, if it doesn't have that we get the very odd situation where the first row of data is used as the field names, let's pull in a CSV file that shows this then look at how to fix it.

A companion to the prescribing data we have already imported is the list of doctors' practice addresses that show geographically where each prescription was issued. This is available from the same source so let's import that too. There is actually one of these files per month, but we will just take the latest one as it doesn't change much. This also assumes you have the HTTPS store configured as above.

```
const addressurl =
'https://files.digital.nhs.uk/9E/27E72D/T201912ADDR%20BNFT.csv'
const dbname = 'nhs'
const virtualcollectionname = 'rawaddresshttp'
const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage
const virtualdb = fedconf.databases.find(db => db.name === dbname)

if (virtualdb) {
       const collection = virtualdb.collections.find(c =>
                            c.name === virtualcollectionname)
     if (collection) {
        collection.dataSources = [{ storeName: 'https',
                                       urls: [addressurl] }]
      } else {
        virtualdb.collections.push({ name: virtualcollectionname,
                              dataSources: [{ storeName: 'https',
                                           urls: [addressurl] }] })
   }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

```
ADF> console.log(fedconf)
{
  stores: [
    {
      name: 's3bucket',
      provider: 's3',
      region: 'eu-west-1',
      delimiter: '/',
      bucket: 'jpage-datalake'
    },
    { name: 'https', provider: 'http' }
  ],
  databases: [
    {
      name: 'nhs',
      collections: [
        {
          name: 'rawdata',
          dataSources: [ { storeName: 's3bucket', path: '/' } ]
        },
        {
          name: 'rawcsvprescribingdata',
          dataSources: [ { storeName: 's3bucket', path: '/rawdata/*' } ]
        },
        {
          name: 'rawaddresshttp',
          dataSources: [
            {
              storeName: 'https',
              urls: [

'https://files.digital.nhs.uk/9E/27E72D/T201912ADDR%20BNFT.csv'
              ]
            }
          ]
        }
      ],
      views: [
        {
          name: 'cleanrawdata',
          source: 'rawcsvprescribingdata',
          pipeline:
'[{"$replaceRoot":{"newRoot":{"$arrayToObject":{"$filter":{"input":{"$map
":{"input":{"$objectToArray":"$$ROOT"},"as":"field","in":{"k":{"$toLower"
```

```
:{"$replaceAll":{"input":"$$field.k","find":"
","replacement":""}},"v":{"$trim":{"input":"$$field.v"}}}}},"as":"field"
,"cond":{"$ne":["$$field.k",""]}}}}}},{"$set":{"actcost":{"$convert":{"in
put":"$actcost","to":"double","onError":0,"onNull":0}},"items":{"$convert
":{"input":"$items","to":"double","onError":0,"onNull":0}},"nic":{"$conve
rt":{"input":"$nic","to":"double","onError":0,"onNull":0}},"quantity":{"$
convert":{"input":"$quantity","to":"double","onError":0,"onNull":0}}}}]'
        }
      ]
    }
  ]
}
ADF> adminDB.runCommand({ storageSetConfig: fedconf })
{ ok: 1 }
ADF>
```

This data does not have a header line as we can see when we look at a record from it. The field names themselves are the first address in the data. We can resolve this with a view, but even better, we can do this when we copy the data from its HTTP source to S3 (or if it was already in S3, we could read it and write to a new file or files in S3 with an appropriate header line).

The raw CSV here looks like this, including dot characters in the first line "DR.CHAUDHURY'S PRACTICE" and "CENTRAL P.C.C LONG ROAD" which Data Federation interprets as nested objects. We will need to do some work to clean this up.

```
201912,F81740,CHAUDHURY SURGERY,DR. CHAUDHURY'S PRACTICE,CENTRAL P.C.C
LONG ROAD,CANVEY ISLAND,ESSEX,SS8 0JA

201912,F81222,QUEENS PARK SURGERY,QUEENSPARK SURGERY,24 THE
PANTILES,BILLERICAY,ESSEX,CM12 0UA

201912,B83661,MOOR PARK MEDICAL PRACTICE,THE BLUEBELL BUILDING,BARKEREND
HEALTH CENTRE,BRADFORD,N/A,BD3 8QH
```

```
ADF> show tables
cleanrawdata            [view]
rawaddresshttp
rawcsvprescribingdata
rawdata
ADF> db.rawaddresshttp.findOne()
{
  '201912': '201912',
  'CANVEY ISLAND': 'BILLERICAY',
  'CENTRAL P': { C: { 'C  LONG ROAD': '24 THE PANTILES' } },
  'CHAUDHURY SURGERY': 'QUEENS PARK SURGERY',
  DR: { " CHAUDHURY'S PRACTICE": 'QUEENSPARK SURGERY' },
  ESSEX: 'ESSEX',
  F81740: 'F81222',
  'SS8 0JA': 'CM12 0UA'
}
ADF>
```

addheaderline.js

```
// First we need to look at the data and establish the column names we want to use */
// First let's fix the issue where the first line having a . in it
//"Dr. Chaudray's practise"  Makes the field be  DR: {" Chaurdray's Practise"}

// Let's do this Explicitly, Putting in the Hash character so we can
// convert back to a dot when we Add this first row back in as data
// Detecting bad fields automatically is more complicated so we will name them here

const badFieldNames = ['DR. CHAUDHURY\'S PRACTICE', 'CENTRAL P.C.C  LONG ROAD']

const fieldnamefixes = {}
for (const badField of badFieldNames) {
 const safeFieldName = badField.replace(/\./g, '#')
 fieldnamefixes[safeFieldName] = `$${badField}`
 // And remove the old one
 fieldPrefix = badField.split('.')[0]
 fieldnamefixes[fieldPrefix] = '$$REMOVE'
}
const fixDottedFieldname = { $set: fieldnamefixes }

// Once we fix the fieldnames, grab the first document
// This tells us the field names, which we can then map and also has the data for the first
// Row which we will need to blend back in

const firstDocument = db.rawaddresshttp.aggregate([{ $limit: 1 }, fixDottedFieldname]).next()
console.log(firstDocument)
```

```
// Make this order match the order in that first document (or define it as a map)
const columns = ['period', 'town', 'surgery', 'county', '_id', 'postcode', 'name', 'street']
let newcolumnindex = 0
const projection = {}

const missingFirstRecord = {}
for (const field in firstDocument) {
 console.log(field)
 const colname = columns[newcolumnindex++]
 projection[colname] = `$${field}`
 missingFirstRecord[colname] = field.replace(/#/g, '.') // Put the dots back
}

const renameFields = { $project: projection }
console.log(renameFields)
console.log(missingFirstRecord)

/* Now put it all together, for each thign in the CSV fix the field names
map to new field names and write it out, also include our other record explicity at the start
using $documents and $unionWith */

const addFirstRow = { $documents: [missingFirstRecord] }
const transformcsv = { $unionWith: { coll: 'rawaddresshttp', pipeline: [fixDottedFieldname,
renameFields] } }

const writeCleanedCSVToS3 = {
   $out: {
     s3: {
       bucket: 'jpage-datalake',
       region: 'eu-west-1',
       filename: `address`,
       format: { name: 'csv', maxFileSize: '2100MB' },
       errorMode: 'stop'
     }
   }
 }
const pipeline = [addFirstRow,transformcsv,writeCleanedCSVToS3]
console.time()
db.rawaddresshttp.aggregate(pipeline)
console.timeEnd()
```

This will read from HTTP, clean up the data, and write it out S3 as address.csv. We can attach that S3 CSV file as a collection (s3address) to get access to it and see the new format.

```
const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand( { "storageGetConfig" : 1 } ).storage

const dbname = 'nhs'
const collectionname = 's3address'
```

```
const dataSources =[{ storeName: 's3bucket', path: '/address*' }]

const virtualdb = fedconf.databases.find( db => db.name==dbname)

if(virtualdb) {
    //Add or replace in array
    collection = virtualdb.collections.find(c => c.name==collectionname)
    if(collection) {
       collection.dataSources = dataSources
    } else {
       virtualdb.collections.push({name:collectionname,dataSources})
    }
}

adminDB.runCommand( { "storageSetConfig" : fedconf } )
```

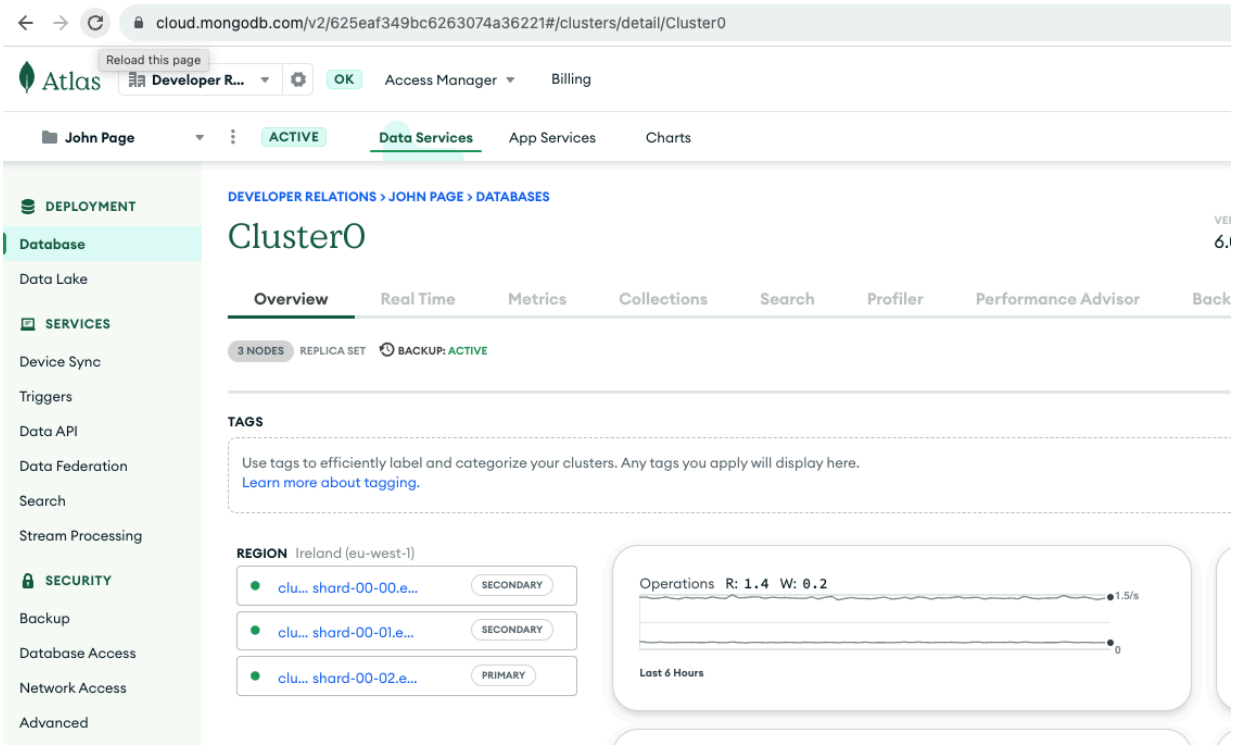
```
ADF> adminDB.runCommand( { "storageSetConfig" : fedconf } )
ADF> { ok: 1 }
ADF> db.s3address.findOne()
{
  _id: 'F81740',
  county: 'ESSEX',
  name: "DR. CHAUDHURY'S PRACTICE",
  period: '201912',
  postcode: 'SS8 0JA',
  street: 'CENTRAL P.C.C  LONG ROAD',
  surgery: 'CHAUDHURY SURGERY',
  town: 'CANVEY ISLAND'
}
```

Attach an Atlas cluster to Federation

We have worked with HTTP- and S3-based data until now, but the other main type of data we need to think about is Atlas clusters. You can attach multiple

Atlas clusters together using Federation, allowing you to query and aggregate over them, adding multiple collections from different databases on one or more clusters to a single database to allow you to perform tasks such as cross database joins.

Adding a collection from an Atlas cluster is actually very simple. You need to know the name of your existing Atlas cluster and also your Atlas Project Id. The simplest way to get both of these is to click on the cluster in the Atlas UI and then take them from the URL. When you see a screen like this, the id is in the URL after v2. It's the long hex string and the final part of the URL. In this case, VOSA is the name of the cluster. Yours might be Cluster0 ot AtlasCluster, as those are defaults.



In the image above with the URL…

https://cloud.mongodb.com/v2/625eaf349bc6263074a36221#/clusters/detail/Cluster0

… the cluster name is **Cluster0** and the cluster id is 625eaf349bc6263074a36221.

To add a collection from this cluster, we use code like this. Note your cluster name is likely to be Cluster0 or AtlasCluster but you may have named it something else.

```
const clusterName = 'Cluster0' // Your is probably  Cluster0 or
AtlasCluster not VOSA
const projectId = '625eaf349bc6263074a36221'
const storeName = 'atlas'
const provider = 'atlas'

const virtualdbName = 'nhs'
const virtualCollectionName = 'addresses_atlas'

const atlasDatabase = 'nhs'
const atlasCollection = 'practise_addresses'


// Add the Cluster to data source if it's not there

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage
if (!fedconf.stores.find(st => st.name === storeName)) {
 const atlasstore = { name: storeName, provider, clusterName, projectId }
 fedconf.stores.push(atlasstore)
}

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 if (!virtualdb.collections.find(c => c.name === virtualCollectionName))
{
   const virtualcollection = {
     name: virtualCollectionName,
     dataSources: [{ storeName, database: atlasDatabase , collection:
atlasCollection }]
   }
   virtualdb.collections.push(virtualcollection)
 }
}

console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

We now have the collection in the virtual nhs database addresses_atlas. We can add an Atlas collection even before it exists as we have done here. When we query it, we will find no data if either it's empty or does not exist. In the next recipe, we will use Data Federation to import data from S3 into this collection, creating it in the process.

```
ADF> show tables
addresses_atlas
cleanrawdata            [view]
rawaddresshttp
rawcsvprescribingdata
rawdata
s3address
ADF> db.addresses_atlas.find()
ADF>
```

Import data from S3 to Atlas

If we have data in S3 (or available via HTTPS), we can use $out to import it into a live Atlas cluster visible inside Data Federation as well as make it accessible to applications connecting to the federated Atlas cluster directly. In this recipe, we are going to import the address data we cleaned up in an earlier exercise into an Atlas cluster.

When we cleaned up the data, we also mapped the Practices ID field to a field called _id. Inside MongoDB collections _id is the unique identifier or primary key for each record and therefore, it always has an index. Even without building any additional indexes, it is quick to look up things by _id in an Atlas cluster. By loading it into Atlas rather than keeping it in S3, we will also be able to efficiently join address information to the data in S3 in a later recipe.

We write to Atlas from Federation using a special form of $out. You cannot simply use $out to a collection as you would in MongoDB because a virtual

collection could have multiple sources behind it. Therefore, to read from the addresses in S3 and write to Atlas, we use code like this. We do not need to have added Atlas as a store to write data to it — only to read from it.

```
const clusterName = 'Cluster0 // Your is probably named Cluster0 or AtlasCluster
const projectId = '625eaf349bc6263074a36221'
const atlasDatabase = 'nhs'
const atlasCollection = 'practise_addresses'

const outToAtlas = { $out: { atlas: { projectId, clusterName, db: atlasDatabase,
coll: atlasCollection } } }

db.s3address.aggregate(outToAtlas)
```

```
ADF> show tables
addresses_atlas
cleanrawdata            [view]
rawaddresshttp
rawcsvprescribingdata
rawdata
s3address
ADF> db.addresses_atlas.findOne()
{
  _id: 'F81740',
  county: 'ESSEX',
  name: "DR. CHAUDHURY'S PRACTICE",
  period: '201912',
  postcode: 'SS8 0JA',
  street: 'CENTRAL P.C.C  LONG ROAD',
  surgery: 'CHAUDHURY SURGERY',
  town: 'CANVEY ISLAND'
}
```

To demonstrate further aspects of Data Federation, we will need to import some of our prescribing data into Atlas. For speed and to limit how much Atlas

disk we need, we will limit the number of records we bring in to 10 million in this exercise.

The code below takes three minutes and 55 seconds to load 10 million rows from CSV held in S3 into Atlas.

```
const clusterName = 'Cluster0'
const projectId = '625eaf349bc6263074a36221'
const atlasDatabase = 'nhs'
const atlasCollection = 'prescriptions'

const outToAtlas = { $out: { atlas: { projectId, clusterName, db:
atlasDatabase, coll: atlasCollection } } }
limitQuantity = { $limit: 10_000_000 }
console.time()
db.cleanrawdata.aggregate([limitQuantity,outToAtlas])
console.timeEnd()
```

We also need to make that newly created collection visible in Federation. Here we will make it visible as a collection called "prescriptions_atlas."

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'prescriptions_atlas'

const atlasDatabase = 'nhs'
const atlasCollection = 'prescriptions'


// Add the Cluster to data source if it's not there

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 if (!virtualdb.collections.find(c => c.name === virtualCollectionName)) {
   const virtualcollection = {
     name: virtualCollectionName,
     dataSources: [{ storeName, database: atlasDatabase , collection:
atlasCollection }]
   }
```

```
    virtualdb.collections.push(virtualcollection)
  }
}

console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

```
ADF> db.prescriptions_atlas.findOne()
{
  _id: ObjectId("65097082f31f51700b4b3ab3"),
  sha: 'Q35',
  actcost: 193.59,
  bnfcode: '0601012X0BBABAB',
  bnfname: 'Ins Levemir_FlexPen 100u/ml 3ml Pf Pen',
  items: 4,
  nic: 210,
  pct: '5P2',
  period: '201106',
  practice: 'E81036',
  quantity: 25
}
ADF> db.prescriptions_atlas.countDocuments()
10000000
ADF>
```

Export data from Atlas to S3

We can read data from Atlas and write it to S3 using a similar technique. We already read from an HTTPS source and wrote to S3. The process is exactly the same when we want to read from Atlas and write to S3. These exercises assume you imported 10 million rows in the previous exercise.

Now we want to export information about a specific drug to S3 so that data can be processed by our analysts. We just need to use $match to select the

data and then $out with the S3 option to write the data out to S3. This aggregation works out what drug is the most prescribed using $sortByCount and then exports all prescripts for that drug back to a CSV file in S3. We could also have done this filtering directly from our S3-based collection into S3.

It turns out the most common item is latex gloves! It takes 19 seconds to work that out (there are no indexes to help) and then another two seconds to write those to S3, for a total of 20.4 seconds.

```
console.time()
const mostPrescribed =
db.prescriptions_atlas.aggregate({$sortByCount:"$bnfname"}).next()
console.log(mostPrescribed)
console.timeLog()


const writeToS3 = {
   $out: {
     s3: {
       bucket: 'jpage-datalake',
       region: 'eu-west-1',
       filename: 'most_prescribed/drug',
       format: { name: 'csv', maxFileSize: '10MB' },
       errorMode: 'stop'
     }
   }
 }

const selectRecords = { $match : { bnfName: mostPrescribed._id}}
db.prescriptions_atlas.aggregate[{selectRecords, writeToS3}]
console.timeEnd()
```

```
ADF> const selectRecords = { $match : { bnfName: mostPrescribed._id}}
ADF> db.prescriptions_atlas.aggregate[{selectRecords, writeToS3}]
ADF> console.timeEnd()
%s: %s default 20.443s
ADF>
```

Move data to an S3 Archive

What if we want to save on running costs and archive older, less frequently used data in S3? Every day, for example, we will copy all data created more than a year ago to S3 and then delete it from the main cluster. There are some subtleties and edge cases to doing this correctly which this recipe explains.

When writing to S3, Data Federation always overwrites existing data with the same path. This means if you are using a scheduled job to write data, it is critical that you supply a sub path to ensure the archive job doesn't overwrite older data. We will illustrate this by archiving one million of our prescription records from Atlas twice.

Normally you would include a $match to select the records you want to delete — for example, likely based on a date field or on a field with an ObjectId (ObjectId values include the date they were created). So you might have an index on create_date and use something like this:

```
const oneYearAgo = new Date();
oneYearAgo.setFullYear(oneYearAgo.getFullYear() - 1);
console.log(oneYearAgo)

chooseRecordsToAtrchive = { $match : { create_date: { $lt: oneYearAgo }}}
```

Our example prescription data has a period field and a string with the year and month which we could use but when we copied over 10 million records, they likely are all from the same month. There is a *lot* of data. Therefore, we will simply archive the first million records and then a second million. This is helpful as there is no specific query that selects for those records, allowing us to illustrate how to safely move data.

First, we want to move one million records to S3. We will put them into a folder called online_archive and store them as compressed BSON, as that is a compact format that preserves all the MongoDB data types. We use $limit to make it one million and we also use a sub directory with a random value. This allows us to keep the data separate and not overwrite existing archives and also refer specifically to the files in this archive.

First we make
"s3://bucketname/online_archive/prescriptions/batchid/prescriptions" visible

as a virtual collection called prescriptions_archive. We are doing something new here in that we are also telling it that one of the parts of the pathname should be treated as a virtual field we can include in any queries against this data. We will talk more about data partitioning techniques later.

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'prescriptions_archive'
const storeName = 's3bucket'

//Adding that part of the path should be treated as a virtual field
//For query purposes with {fieldname type} syntax

const s3Path = '/onlineArchive/prescriptions/{archive_batchid
objectid}/prescription'

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 let virtualcollection = virtualdb.collections.find(c => c.name ===
virtualCollectionName)
 if (virtualcollection) {
  virtualcollection.dataSources = [ { storeName, path: s3Path} ]
} else {
   virtualcollection = {
     name: virtualCollectionName,
     dataSources: [ { storeName, path: s3Path} ]
   }
   virtualdb.collections.push(virtualcollection)
 }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

Now, we will write out our first million documents using a unique batch id. This will also print out the batch ID. We should take a note of it as we will need it to remove those documents from the cluster later.

```
const firstOneMillion = { $limit: 1_000_000 }
const batchId = new ObjectId()
```

```
const path = `onlineArchive/prescriptions/${batchId}/prescription`

const writeCompressedBSON = {
 $out: {
   s3: {
     bucket: 'jpage-datalake',
     region: 'eu-west-1',
     filename: path,
     format: { name: 'bson.gz', maxFileSize: '5MB' },
     errorMode: 'stop'
   }
 }
}

db.prescriptions_atlas.aggregate([firstOneMillion,writeCompressedBSON])
print(`TAKE A NOTE OF THIS : ${batchId}`)
```

My batch id was 509764dc891f3214f2d1458. We can verify the data we moved over by querying for one document and looking to see the value of the archive_batchid field as below. We can also count and see that we did indeed copy over one million documents.

```
ADF> db.prescriptions_archive.findOne()
{
  _id: ObjectId("65097088f31f51700b51211e"),
  sha: 'Q30',
  actcost: 682.05,
  bnfcode: '0407020C0AAADAD',
  bnfname: 'Codeine Phos_Tab 15mg',
  items: 126,
  nic: 735.69,
  pct: '5D7',
  period: '201111',
  practice: 'A86012',
  quantity: 8873,
  archive_batchid: ObjectId("6509764dc891f3214f2d1458")
}
ADF> db.prescriptions_archive.countDocuments()
1000000
ADF>
```

Now, we need to remove those one million records from the Atlas cluster. If we connect directly to Atlas, we might use DeleteMany for this. But depending on how we selected these records for archiving, how can we be sure we will get exactly the same results we did when archiving? What if, during this process, a new record was added with an old date? Or what if we were archiving customers who hadn't used our system for a year based on last_used_date but then one of them just used us again?

Because of this potential race condition, what we want to do is explicitly remove the records we just moved to the archive. We cannot use updateOne or deleteOne from Data Federation. We could use $out to read the Atlas collection and overwrite it with a new copy that doesn't have the records we copied, but that is both difficult as we would need a huge $match clause (or many $match clauses) and also, we would lose changes that happened to the source collection whilst we were creating our filtered one.

The solution to this happens in two stages. First, we will use $merge to update the Atlas collection. We cannot delete documents using $merge but we can update the content, so we will use it to replace records in Atlas with ones having one field archive_date. We will then take advantage of the "Time to Live" index feature in Atlas to automatically delete any record as soon as an archive_date field appears in it.

The following code looks for a specific value of archive_batchid in the S3 archive and rewrites those documents in the live Atlas collection with only an archive_date field. The archive_batchid value is the one you took note of in the previous example.

```
const clusterName = 'Cluster0' // Your is probably Cluster0 or
AtlasCluster
const projectId = '625eaf349bc6263074a36221'
const atlasDatabase = 'nhs'
const atlasCollection = 'prescriptions'

const archive_batchid = ObjectId('6509764dc891f3214f2d1458')
const findBatch = { $match: { archive_batchid } }

const createDeleteRecord = { $project: { archive_date: '$$NOW' } }

const mergeIntoLive = {
```

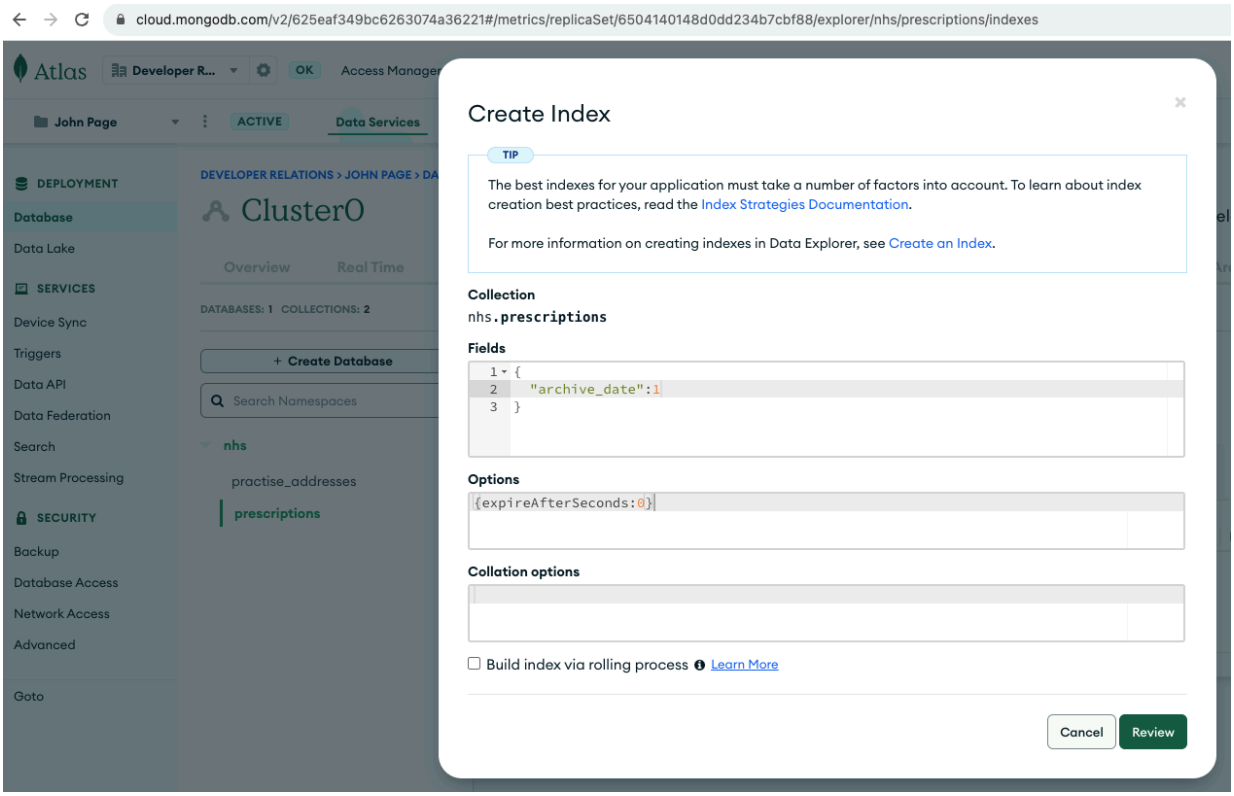
```
 $merge: {
   into: {
     atlas: {
       projectId,
       clusterName,
       db: atlasDatabase,
       coll: atlasCollection
     }
   },
   on: '_id',
   whenMatched: 'replace', // Replace with this version
   whenNotMatched: 'discard' // Already deleted
 }
}
console.time()
db.prescriptions_archive.aggregate([findBatch,
createDeleteRecord,mergeIntoLive])
console.timeEnd()
```

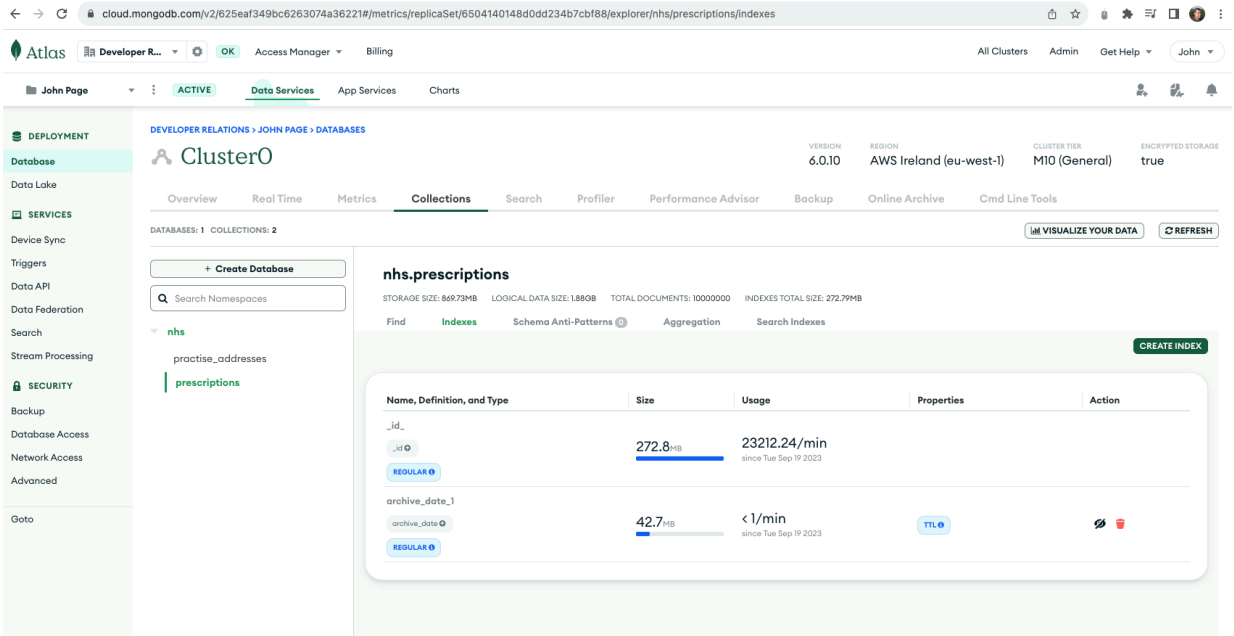
This takes about one minute and 40 seconds to update one million documents on the lowest dedicated Atlas tier (an M10) when nothing else is happening. We can see what it has done by looking at our 10 million document Atlas collection and finding how many documents are flagged for deletion.

```
ADF> console.time()
ADF> db.prescriptions_archive.aggregate([findBatch, createDeleteRecord,mergeIntoLive])
ADF> console.timeEnd()
%s: %s default 1:38.793 (m:ss.mmm)
ADF> db.prescriptions_atlas.countDocuments({archive_date:{$lt: new Date()}})
1000000
ADF>
```

Finally, we can configure Atlas to remove these documents automatically from the Atlas cluster as they appear. For this, we create a time to live index with a time of 0. Time to live indexes define how many seconds after a given date a record should be retained before auto deletion. You can set them to delete 24

hours after creation, for example, but in our case, we explicitly set the field. As soon as the field is written, we want them to expire. You can read about TTL indexes and set them programmatically or using the Atlas console, like so.





Deleting expired documents happens in the background. If we check every 10 seconds or so, we can see these documents we move to the online archive being deleted.

```
ADF> db.prescriptions_atlas.countDocuments({archive_date:{$lt: new Date()}})
1000000
ADF> db.prescriptions_atlas.countDocuments({archive_date:{$lt: new Date()}})
721397
ADF> db.prescriptions_atlas.countDocuments({archive_date:{$lt: new Date()}})
429683
ADF> db.prescriptions_atlas.countDocuments({archive_date:{$lt: new Date()}})
193584
ADF> db.prescriptions_atlas.countDocuments({archive_date:{$lt: new Date()}})
2435
ADF> db.prescriptions_atlas.countDocuments({archive_date:{$lt: new Date()}})
0
ADF> db.prescriptions_atlas.countDocuments({})
9000000
ADF>
```

Building a virtual collection over multiple sources

Until now, we have created virtual collections looking at a single data source. We have two collections — *prescriptions_atlas* and *prescriptions_archive* — as well as the huge *rawcsvprescribingdata* virtual collection we will look at later. What if we wanted to combine Atlas and archive together into a single virtual collection that we can use to query both simultaneously. We do this by adding multiple data sources to the virtual collection definition. Here we create a prescriptions_all virtual collection which includes both Atlas and the archived data in S3.

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'prescriptions_all'

const s3StoreName = 's3bucket'
const s3Path = '/onlineArchive/prescriptions/{archive_batchid
objectid}/prescription'

const atlasStoreName = 'atlas'
const atlasDatabase = 'nhs'
const atlasCollection = 'prescriptions'
const dataSources = [
 { storeName:s3StoreName, path: s3Path},
 { storeName: atlasStoreName,database:  atlasDatabase , collection:
atlasCollection }
]

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 let virtualcollection = virtualdb.collections.find(c => c.name ===
virtualCollectionName)
 if (virtualcollection) {
  virtualcollection.dataSources = dataSources
} else {
   virtualcollection = {
     name: virtualCollectionName,
     dataSources   }
   virtualdb.collections.push(virtualcollection)
 }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

Now, we have another virtual collection prescriptions_all that has both Atlas and S3 data in it.

```
ADF> db.prescriptions_atlas.countDocuments()
9000000
ADF> db.prescriptions_archive.countDocuments()
```

```
1000000
ADF> db.prescriptions_all.countDocuments()
10000000
ADF>
```

There is no simple solution to ensuring you do not get duplicate results whilst data is being archived. As archiving is a multi step process, it is possible a document has been written to S3 but not yet removed from live. There are a number of ways to mitigate this but they are dependent on your exact use case. For example, you could add all the S3 archives explicitly to an archive collection rather than using a higher level path to see them all. If you do this, though, you may have a period when deleting from live where a record is available in neither live nor archive.

You could also use $group on _id and $first when reading the data to remove duplicates. This works well for small sets, like hundreds of results, but would take a long time for millions of documents.

Handling failure and idempotency

When using Data Federation to move and copy data, the data we are writing is not written transactionally. In the online to archive example above, we could fail when reading from Atlas and writing to S3, leaving some data copied to S3 — for example, if the Atlas instance goes down or our code running the operation fails. (There is an option to run it in the background so our appserver/shell does not need to be there for the whole process.) We might copy to S3 and then fail before flagging for deletion or during flagging for deletion.

To make this all very robust, we need to consider making operations idempotent and using flag files to identify things having completed.

An idempotent operation is one you can perform again and again and get the same outcome. When we write out to S3 with a given path, it will overwrite all existing data with the same path. This is good as it makes these operations idempotent. If it fails part way, we can run it again. We do need to be conscious of this when we want a separate set of data to be written.

If we want to be sure that, for example, a write operation has completed, then in our code, we should first write flag files using $out (and take advantage of the fact it overwrites data).

Suppose we are doing the online archive process above. It might stop at any time, and we need to be able to resume or restart it. Also, assume we are using a query to identify the data we want to archive from Atlas.

We should first record that we are starting an archive process and what data we are archiving, so before copying data from Atlas to S3, we should write a file to S3 saying what we are doing. We can use $documents and $out to record that we are copying data older than date X from Atlas to S3, and record the batchid we are using.

We write this to its own directory, including the batchid — perhaps in the filename — and configure this as a virtual collection where we can query the state of all archive jobs.

When our code completes, we can use $documents and $out again to say now we are using $merge to remove those files from Atlas.

When that completes, we can overwrite this file again to say that we are done.

If anything fails, we can query the archive_jobs virtual collection and see what failed at what state and run it again to ensure we get all data both archived and removed from live, even through a failure.

As a quick and simplified example, imagine we are copying data from a cluster to S3 every day and we need to record the last day we successfully completed copying.

First, imagine we have a production collection in a MongoDB Atlas cluster. We can attach this collection as a virtual collection in Data Federation. Here we will attach one and also run an aggregation to generate some data in it, for demonstration purposes. We will use a database called “production” and a collection called “events.”

```
const clusterName = 'Cluster0'
const projectId = '625eaf349bc6263074a36221'
const storeName = 'atlas'
```

```
const provider = 'atlas'

const virtualdbName = 'nhs'
const virtualCollectionName = 'appointments'

const atlasDatabase = 'nhs'
const atlasCollection = 'appointments'


// Add the Cluster to data source if it's not there

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

if (!fedconf.stores.find(st => st.name === storeName)) {
 const atlasstore = { name: storeName, provider, clusterName, projectId }
 fedconf.stores.push(atlasstore)
}

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)

if (virtualdb) {
 if (!virtualdb.collections.find(c => c.name === virtualCollectionName))
{
   const virtualcollection = {
     name: virtualCollectionName,
     dataSources: [{ storeName, database: atlasDatabase , collection:
atlasCollection }]
   }
   virtualdb.collections.push(virtualcollection)
 }
} else {
  console.log("You need to add the Virtual Database first")
}

console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

We can then create some "appointment" records to use, for demonstration purposes. The content of these does not really matter. What we are interested in is the the _id field which will contain a unique, incrementing appointment number.

```
const createOne = { $documents : [ {} ] }

//Add an array and unwind to make many copies of each
const nDups = 100

const duplicate  = [ { $set : { a : {$range:[0,nDups]}}}, {$unwind : "$a"}
]
const duplicate2  = [ { $set : { b : {$range:[0,nDups]}}}, {$unwind :
"$b"} ]

//Take a and b and make a numeric  _id

const _id = {$add : [ "$b",{$multiply:["$a",nDups]}]}
const patientId = {$floor:{$multiply:[ {$rand:{}},100_000_000]}}
const doctorId = {$floor:{$multiply:[ {$rand:{}},500_000]}}
const locationId = {$floor:{$multiply:[ {$rand:{}},8_000]}}
const now= new Date()
const apptDate = { $dateAdd: {
                                        startDate: new Date(),
                                        unit: "day",
                                        amount : {$floor:{$multiply:[
{$rand:{}} ,180 ]}}}}



const rewrite = { $project : { _id ,patientId,doctorId,apptDate,locationId
}}

 const clusterName = 'Cluster0'
 const projectId = '625eaf349bc6263074a36221'
 const storeName = 'atlas'
 const provider = 'atlas'
 const atlasDatabase = 'nhs'
 const atlasCollection = 'appointments'

const writeToCluster = {
  $merge: {
    into: {
      atlas: {
        projectId,
        clusterName,
        db: atlasDatabase,
```

```
        coll: atlasCollection
      }
    }
  }
 }

//Note no collection as we are creating data
db.aggregate([createOne,...duplicate,...duplicate2,rewrite,writeToCluster]
)
```

Now, we want to copy these to files in S3. We will do 1,000 at a time in _id order. (We could also do it bydate or in another order.) Importantly, also record in S3 what the highest number copied was. That way, in the event of a problem copying to S3, we know we have not completed copying that batch and can start again. Essentially, code that every time it is run, it will copy 1,000 records.

First, we add a new virtual collection in S3 that we are archiving the data too. Each batch has its own directory but also, there is a directory `"/onlineArchive/appointments/latest/appointment"` with a single entry that has a single document showing what the last batch we completed contained.

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'appointments_archive'

const s3StoreName = 's3bucket'
const s3Path = '/onlineArchive/appointments/{batchid string}/appointment'

const dataSources = [ { storeName:s3StoreName, path: s3Path} ]

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)

if (virtualdb) {
  let virtualcollection = virtualdb.collections.find(c => c.name ===
virtualCollectionName)
  if (virtualcollection) {
   virtualcollection.dataSources = dataSources
```

```
} else {
   virtualcollection = {
     name: virtualCollectionName,
     dataSources   }
   virtualdb.collections.push(virtualcollection)
 }
} else {  console.log("You need to add the Virtual Database first") }

console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

The code below reads the document showing where to start copying. It copies 1,000 documents from that point and then updates the "where to start" file on success.

```
const nToCopy = 1000
const getStartingPoint =  db.appointments_archive.findOne({batchid:
"last"})
let startFrom = 0


if(getStartingPoint)  { startFrom = getStartingPoint.startFrom }
print(`Copying {$nToCopy} from ${startFrom}`)

 // With dates it's more likely you use a hard bound than a specific
number to copy
// So we copy from and to

const copyTo = startFrom + nToCopy
const getDocsToCopy = { $match : { _id : { $gte: startFrom, $lt: copyTo
}}}

//Used to put this batch in it's own directory to avoid overwriting

 const addBatchIdToEach = { $set : { batchid: startFrom} }


 const controlFile = { batchid: "last", startFrom: copyTo }
 const addControlFile = {$unionWith: { pipeline:[{$documents :
 [controlFile]}]}}
```

```
const path = {"$concat":
["onlineArchive/appointments/",{$toString:"$batchid"},"/appointment"]}

const writeCompressedBSON = {
 $out: {
   s3: {
     bucket: 'jpage-datalake',
     region: 'eu-west-1',
     filename: path,
     format: { name: 'bson.gz', maxFileSize: '20MB' },
     errorMode: 'stop'
   }
 }
}



//

db.appointments.aggregate([getDocsToCopy,addBatchIdToEach,addControlFile,w
riteCompressedBSON])
```

Viewing the query log

Data Federation keeps a log of all operations run which can be useful when debugging and understanding what is going on. You can view this by running an aggregation starting with the stage $queryHistory directly against the admin database. Note that aggregation is against the database admin itself, not against a collection in it.

```
const  adminDB = db.getSiblingDB('admin')
adminDB.aggregate([{$queryHistory:{allUsers:true}},{$sort:{_id:-1}}]
)
```

Records look like this one — in this case, showing us an error message that might be useful in debugging why a query did little or nothing.

```
  {
    _id: ObjectId("650b21d41737e100bd1efa6a"),
    query: [
      {
        '$currentOp': { allUsers: true, idleConnections: false,
truncateOps: false }
      },
      { '$match': {} }
    ],
    appName: 'mongosh 1.10.6',
    user: 'admin.jlp',
    db: 'admin',
    collection: '',
    opid: ObjectId("1786a93915b4e7a427839847"),
    startTime: ISODate("2023-09-20T16:46:12.574Z"),
    background: false,
    endTime: ISODate("2023-09-20T16:46:13.053Z"),
    error: "failed parsing stage: unrecognized option 'truncateOps'
 in $currentOp stage",
    ok: 0
  }
]
Type "it" for more
ADF>
```

Monitoring ongoing operations

We can use **db.currentOp()** to get the status of all current operations and see progress as an absolute value, although not as a percentage of the total. We do this like so — use Control C when you want to stop it. As long as the work done keeps increasing, the job is still running.

```
while(true) {
  const ops =  db.currentOp().inprog
  for(const op of ops) {
    print(`${op.ns} - ${op.msg}`);
```

```
  }
  sleep(60000)
}
```

The output looks like this:

```
AtlasDataFederation test> while(true) {
...   const ops =  db.currentOp().inprog
...   for(const op of ops) {
...     print(`${op.ns} - ${op.msg}`);
...   }
...   sleep(60000)
... }
nhs.cleanrawdata - work done: 3292217345
admin.$cmd.aggregate - work done: 0
nhs.cleanrawdata - work done: 3319781670
admin.$cmd.aggregate - work done: 0
nhs.cleanrawdata - work done: 3347081989
admin.$cmd.aggregate - work done: 0
```

Understanding the cost and behavior of a query

As mentioned at the start, when using Data Federation, queries are charged effectively by how much data they read, write, and return, with the total of that being charged at $5 per TB of data. In some cases, as discussed below, the quantity of data scanned can also have a significant impact on the query performance. It is therefore useful to be able to determine the cost and efficiency of a query before running it. You can get this information from the "query explain plan" like so. As you can see, our entire data set is quite large.

```
use nhs
db.rawdata.find().explain('queryPlannerExtended')
```

This outputs all the partitions being searched, which could be thousands. We can look just at the stats member to get a summary.

```
use nhs
db.rawdata.find().explain('queryPlannerExtended').stats
```

We can also use this to estimate the cost of a query — at least the reading from data sources part that is charged by MongoDB. In this case, it costs us $0.20 to run a query that scans all 45 GB of compressed raw data in S3.

```
function price(collection,query) {
  const ep = db[collection].find(query)
                           .explain('queryPlannerExtended').stats

  const [all,count,type] = ep.size.match(/([0-9\.]*) ([KMGT])/)

  //Find cost not transfer costs of results

  let dataread = ( count / (1000**('TGMK'.indexOf(type))))

  if(dataread < 0.00001) {
    print("Less than 10MB minimum read");
    dataread = 0.00001;
  }

  const queryCost = 5 *dataread
  print(`Partitions: ${ep.numberOfPartitions} `+
        `size: ${ep.size} cost: $${queryCost.toPrecision(1)}`)
}

price("rawdata",{})
```

Optimizing an S3 source for field-based lookup

S3 is not optimized for reading a single document/row. The least we can read is a single file section from a file from S3. In our NHS prescribing data, we have over one billion rows. The way S3 works, it doesn't really have folders but is a

database of files, with each file having a parent value to simulate folder-based storage. Although this isn't normally important, it is worth understanding that listing the contents of a folder is relatively slow and the API call used to do it can return only 10,000 elements per call, so putting millions of files in one folder is a bad idea.

As we saw above, we can designate parts of the pathname as being virtual fields which allows us to narrow down what folders we list contents of and what files we process. If we really do want to be able to get a single document, or a small number by a key, how do we best organize our data to do that?

The minimum read cost for a query or aggregation in Federation is 10MB of data, so making any file smaller than that won't be any cheaper to read. It might be a little faster but processing a single 10MB file will be quick.

We already added a view to clean the raw data up, so let's also apply that and transform the data to a nice format as well as organize it for better retrieval.

Assume we know our users will want to retrieve data on a specific drug. For some drugs, there will be millions of rows. For others, there may be hundreds or dozens. Let's reorganize the data to allow us to retrieve by a field (the drug name) more efficiently. At the same time, we can add a unique id to each row so we can then look into retrieval by a very high cardinality field like a unique identifier.

In order to write this data out to S3, aggregation will need to sort the data by the bnf code (Drug ID), so this may take some time. We are reading data from S3, so we cannot assist the sorting with an index. The Federation engines will need to sort slightly over one billion rows of data internally.

This is writing out to S3 in compressed BSON files up to 100MB each. The directory path is "`prescriptions/bydrug/<BNF PREFIX>/<BNF CODE>`".

BNF PREFIX is the first six characters of the BNF CODE (Drug ID). This lets us avoid having a single huge directory. As an aside, it also categorises by the type of drug it is. Read more about BNF codes.

We are also adding a large random number to each record in the _id field as in this case, they don't have a high cardinality key in this data set (or rather, it's been redacted when the data was open sourced). But we would like to have one to demonstrate how to quickly find data that has.

This takes 95 minutes to complete for these one billion records.

```
const cleanedview  = db.cleanrawdata // Our view setting the data types

//These aren't guaranteed to be unique but they are nice high cardinality
//So we can use them later for high cardinality tests

bigrandomnumber = {$toLong:{ $multiply:[ {$rand:{}}, 1_000_000_000]}}
//Add a 1 at the start then truncate string to 0 pad
randString = {$substr: [{ $toString: {$add:[ 1_000_000_000,bigrandomnumber]}},1,-1]}

addId = { $set : { _id :randString}}

//We will write out BSON rather than CSV - it has the data converted to binary types so
//we wont have the costs of CSV parsing reading
//We will have the path based on the bnf code (we could use bnf name)
//Infor about BNF codes is here
// https://www.bennett.ox.ac.uk/blog/2017/04/prescribing-data-bnf-codes/

//To avoid huge directories I am adding a virtual field with teh BNF code prefix

const addBNFPrefix = { $set : { bnfprefix : { $substr: [ "$bnfcode",0,6]}}}

const outfilename = {"$concat":
["prescriptions/bydrug/","$bnfprefix","/","$bnfcode","/"]}

const writeCompressedBSON = {
 $out: {
   s3: {
     bucket: 'jpage-datalake',
     region: 'eu-west-1',
     filename: outfilename,
     format: { name: 'bson.gz', maxFileSize: '100MB' },
     errorMode: 'stop'
   }
 }
}

console.time()
cleanedview.aggregate([addId,addBNFPrefix,writeCompressedBSON])
console.timeEnd()
```

Configure this S3 data as a new virtual collection.

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'prescriptions_bydrug'

const s3StoreName = 's3bucket'
const s3Path = 'prescriptions/bydrug/{bnfprefix string}/{bnfcode string}'
```

```
const dataSources = [
 { storeName:s3StoreName, path: s3Path},
 ]

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 let virtualcollection = virtualdb.collections.find(c => c.name ===
virtualCollectionName)
 if (virtualcollection) {
  virtualcollection.dataSources = dataSources
} else {
   virtualcollection = {
     name: virtualCollectionName,
     dataSources   }
   virtualdb.collections.push(virtualcollection)
 }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

Partitioning the data by the bnfcode field allows us to target a query to a subset of the data if we provide the BNF code. If we compare the timing of queries where we do and do not supply the BNF code, we can see they hit a dramatically different number of files and thus have differing costs.

```
function price(collection,query) {

const ep = db[collection].find(query).explain('queryPlannerExtended').stats
const [all,count,type] = ep.size.match(/([0-9\.]*) ([KMGT])/)
//Find cost not transfer costs of results
let dataread = ( count / (1000**('TGMK'.indexOf(type))))
if(dataread < 0.00001) { print("Less than 10MB minimum read");dataread = 0.00001;}

const queryCost = 5 *dataread
print(`Partitions: ${ep.numberOfPartitions} size: ${ep.size} cost: $${queryCost.toPrecision(1)}`)
}

price("prescriptions_bydrug",{ bnfname: 'Co-Magaldrox_Oral Susp 200/175mg/5ml S/F'})

price("prescriptions_bydrug",{ bnfcode: '0101010I0AABIBI'})
```

```
ADF> use nhs
ADF> price("prescriptions_bydrug",{ bnfname: 'Co-Magaldrox_Oral Susp 200/175mg/5ml S/F'})
Partitions: 38817 size: 19.07141580991447 GiB cost: $0.1
ADF>
ADF> price("prescriptions_bydrug",{ bnfcode: '0101010I0AABIBI'})
Less than 10MB minimum read
Partitions: 1 size: 36.935546875 KiB cost: $0.00005
```

We can also compare the performance of these queries. As Data Federation can query multiple files in parallel, the time is not directly proportional to the volume of data. We also cannot time using a simple find() as that retrieves only the first 100 documents, unless we iterate over the cursor. Instead, we will use countDocuments which has to find all of them.

```
ADF> console.time()
ADF>  db.prescriptions_bydrug.countDocuments({ bnfcode: '0101010G0AAAGAG'})
1746
ADF>  console.timeEnd()
%s: %s default 32.186s

ADF>  console.time()
ADF>  db.prescriptions_bydrug.countDocuments({ bnfcode: '0101010G0AAAGAG'})
1746
ADF>  console.timeEnd()
%s: %s default 1.581s
```

We can see that by partitioning out data based on the field we want to query, we can drop the query time from 32 seconds to 1.5 seconds as well as dramatically drop the query cost.

When we partitioned out data to avoid having one huge folder, we created a subfolder based on the first six characters of the BNF code. (The first six characters actually denote what the drug is used for.) This won't change the quantity of data we read but it does reduce the number of folders we need to look at in S3 to find those files. Let's see what impact it has on performance.

```
const bnfcode = '0101010G0AAAGAG'
const bnfprefix = bnfcode.substr(0,6)

console.time()
db.prescriptions_bydrug.countDocuments({ bnfcode})
console.timeEnd()

console.time()
db.prescriptions_bydrug.countDocuments({bnfcode,bnfprefix})
console.timeEnd()
```

```
ADF> const bnfcode = '0101010G0AAAGAG'
ADF> const bnfprefix = bnfcode.substr(0,6)

ADF> console.time()
ADF> db.prescriptions_bydrug.countDocuments({ bnfcode})
1746
ADF> console.timeEnd()
%s: %s default 1.575s

ADF> console.time()
ADF> db.prescriptions_bydrug.countDocuments({bnfcode,bnfprefix})
1746
ADF> console.timeEnd()
%s: %s default 716.051ms
```

Using the *bnfprefix* field, we dropped our queries from 1.5 seconds to 0.7 seconds.

Optimizing an S3 source for time-based queries

We optimized for queries looking at a specific drug — or perhaps a group of drugs — by using a $or clause in the query, but we might also want to look at drugs over a period of time. The simplest way to do this is to put it in a time-based hierarchy — for example /year/month/data.json. The most flexible way to do this is to add fields with aggregation. Here we are reading from the hierarchy and writing out by date. This data doesn't have a date field in it — only a string with year and month. So we will parse that and add a random day field for illustration purposes. Most data would likely have a day field, too. This has to get all of the data and sort it by data before writing it out, so it takes a long time to run. Expect this to take just over two hours to complete.

```
const day = {$add: [ 1, {$toInt: { $multiply: [ 28, {$rand:{}}]}}]}

const year = {$toInt: { $substrCP: [ "$period",0,4]}}
const month = {$toInt: {$substrCP : ["$period",4,2]}}

const dayPadded = {$concat: ["0",{$toString:"$$day"}]}
dayPadded = {$substrCP: [ dayPadded,
                          { $subtract:[{$strLenCP:dayPadded},2]}
                          ,2]} //Last 2 characters


const dateString = {$concat : [{ $substrCP: [ "$period",0,4]},
"-",{$substrCP : ["$period",4,2]},"-",dayPadded]}


const date = { $toDate : dateString }

// Bug where $rand is evaluated multiple times even if it own stage
// so we need to use $let and $replaceRoot
```

```
const addDateParts = { $replaceRoot : { newRoot: {
  $mergeObjects : [
     {$let: {
     vars: { day },
       in: { day:"$$day",month,year, dateString,date }}
     },
     "$$ROOT"] }}}

const cleanedview  = db.cleanrawdata // Our view setting the data types

const outfilename = {"$concat":
["prescriptions/bydate/",{$toString:"$year"},"/",{$toString:"$month"},"/"
,{$toString:"$day"},"/"]}

const writeCompressedBSON = {
 $out: {
   s3: {
     bucket: 'jpage-datalake',
     region: 'eu-west-1',
     filename: outfilename,
     format: { name: 'bson.gz', maxFileSize: '100MB' },
     errorMode: 'stop'
   }
 }
}

console.time()
cleanedview.aggregate([addDateParts,writeCompressedBSON])
console.timeEnd()
```

Add the data source.

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'prescriptions_bydate'

const s3StoreName = 's3bucket'
const s3Path = 'prescriptions/bydate/{year int}/{month int}/{day int}/'

const dataSources = [
 { storeName:s3StoreName, path: s3Path},
 ]
```

```
const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 let virtualcollection = virtualdb.collections
                                  .find(c=>c.name ===
virtualCollectionName)
 if (virtualcollection) {
  virtualcollection.dataSources = dataSources
} else {
   virtualcollection = {
     name: virtualCollectionName,
     dataSources   }
   virtualdb.collections.push(virtualcollection)
 }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

We can do a basic range-based query and query all data, but we can also use partitions by having a function that computes the partitions to query, like so.

```
// Adding Date Ranges to Query

function optimiseDateQuery (fromDate, toDate) {
  const minyear = fromDate.getFullYear()
  const minmonth = fromDate.getMonth() + 1
  const minday = fromDate.getDate()

  const maxyear = toDate.getFullYear()
  const maxmonth = toDate.getMonth() + 1
  const maxday = toDate.getDate()

  const partsOfRange = []

  // First month
  if (minyear === maxyear && minmonth === maxmonth) {
    partsOfRange.push({ day: { $get: minday, $lt: maxday },
                        month: minmonth, year: minyear })
    return partsOfRange
  } else {
```

```
    partsOfRange.push({ day: { $gt: minday }, month: minmonth,
                                year: minyear })
  }

  // Whole months in first part year
  if (minyear === maxyear) {
    partsOfRange.push({ month: { $gte: minmonth, $lte: maxmonth },
                        year: minyear })
  } else {
    partsOfRange.push({ month: { $gte: minmonth }, year: minyear })
  }

  // whole years
  if (maxyear - minyear > 1) {
    partsOfRange.push({ year: { $gt: minyear, $lt: maxyear } })
  }

  // Months in last partial year
  if (maxyear > minyear) {
    partsOfRange.push({ month: { $lt: maxmonth }, year: maxyear })
  }

  // Days of last month (not inclusive)
  partsOfRange.push({ day: { $lt: maxday }, month: maxmonth,
                      year: maxyear })

  return partsOfRange
}

const fromDate = new Date('2016-08-05')
const toDate = new Date('2019-11-04')

simpleDateSearch = { date: { $gte: fromDate, $lt: toDate } }

const rangeQuery = {$or : optimiseDateQuery(fromDate, toDate)}

//Combine the obejcts
rangeQuery.date =  { $gte: fromDate, $lt: toDate }


console.time()
console.log("Unoptimised Query")
console.log(simpleDateSearch)
db.prescriptions_bydate.countDocuments(simpleDateSearch)
console.timeEnd()
```

```
console.time()
console.log("Unoptimised Query")
console.log(rangeQuery)
db.prescriptions_bydate.countDocuments(rangeQuery)
console.timeEnd()
```

```
{
  date: {
    '$gte': ISODate("2016-08-05T00:00:00.000Z"),
    '$lt': ISODate("2019-11-04T00:00:00.000Z")
  }
}
ADF> db.prescriptions_bydate.countDocuments(simpleDateSearch)
285458300
ADF> console.timeEnd()
%s: %s default 21.048s

ADF> console.time()
ADF> console.log(rangeQuery)
{
  '$or': [
    { day: { '$gt': 5 }, month: 8, year: 2016 },
    { month: { '$gte': 8 }, year: 2016 },
    { year: { '$gt': 2016, '$lt': 2019 } },
    { month: { '$lt': 11 }, year: 2019 },
    { day: { '$lt': 4 }, month: 11, year: 2019 }
  ],
  date: {
    '$gte': ISODate("2016-08-05T00:00:00.000Z"),
    '$lt': ISODate("2019-11-04T00:00:00.000Z")
  }
}
ADF> db.prescriptions_bydate.countDocuments(rangeQuery)
```

```
285458300
ADF> console.timeEnd()
%s: %s default 18.536s
```

This seems like a relatively small difference — 21 seconds versus 18.5 seconds. We can use the *price()* function we defined above to compare the work done and therefore the cost of these two queries.

```
ADF> price("prescriptions_bydate",simpleDateSearch)
Partitions: 2800 size: 37.444044074974954 GiB cost: $0.2
ADF> price("prescriptions_bydate",rangeQuery)
Partitions: 812 size: 10.804908664897084 GiB cost: $0.05
```

We see that the second query queries only about 25% of the data and therefore costs $0.05 rather than $0.20. Why is it not four times faster then? The answer is that Data Federation can throw a *lot* of parallel processing at a query to make it as fast as possible and in this case, probably used three times as many CPUs. In the next section, we will look at how we can optimize for speed.

Optimizing an S3 source as a general purpose data lake

A traditional data lake does very little data pre-organization and simply throws parallel compute power at answering questions. Atlas Data Federation can be used in this way, too, to provide a simple source of truth for answering any question. Getting the most out of this, though, is easier if you understand a few principles.

Let's start with the costs. You are charged for the data you read and return. If we take the approach of not optimizing for any query, then you will have a

constant cost per query — currently $5 per TB of data processed. This refers to the S3 file sizes, so using compressed files can save a lot of money at the expense of perhaps slower queries.

Atlas Data Federation can throw a *lot* of CPU cores and processing power at the queries you send. The total available is not published, but we can test and see if there is some useful limit to how much you should split your data. We will start all our rows as a single file and compare the performance of a query over all that data.

For reference, we can compare this to the query done against a database instance, too. A query on an unsharded cluster in MongoDB will be single threaded but also have no per-query charge.

About MongoDB M10 and M20 images burstable CPU

It is possible to test this using an M10, the lowest cost dedicated image on Atlas. To do this, we need to be aware of certain caveats.

*To allow all the data **to fit on an M10**, ensure it is a **new M10 cluster with 128GB of disk** and that you have **disabled auto-scaling** and **all backup**s. Auto-scaling will cause the instance size to grow, at least once as we are pushing it as hard as we can. When backups are enabled, the instance retains 24 hours of transactions in the oplog. As we are filling it 95% with data inside 24 hours, there is no space to store this second copy.*

M10 and M20 images do not offer 100% CPU continuously. They use a system of CPU credits .-An M10 (which has two CPUs) earns 24 credits per hour and a credit is equal to one CPU at 100% for one minute. This is 20% per core, so essentially ,it has CPUs that run continuously at 20% or in bursts of up to 100%. You can "bank" up to 288 minutes of 100% CPU time on an AWS M10 when using it at less than 20%.

When loading data, we will quickly eat up that time. So if you watch, you will load at 80,000 documents a second for a few minutes, and then it drops to 16,000 documents a second for the next 15 hours or so.

When performing performance testing, however, we want to have enough CPU credits to show the performance we would get on an M30 or larger, where the CPUs are the same speed as full burst speed on an M10 and M20.

The aggregation will be single threaded, so if we can estimate the time it takes, we

know how long the cluster must be idle before our test. The test takes about 40 minutes to run, so 40/24 minutes means ***we must allow about two hours of idle time before running the test.***

Writing this data to Atlas creates a collection with 87 GB of compressed data and 28 GB of indexes for 114GB of total storage. The uncompressed data size is over 230 GB. This ***just fits on a fresh Atlas M10*** *with the disk size set to the maximum 128B, with just enough room for the oplog in about 15 hours* ***as long as auto-scaling and backups are disabled.***

```
db=db.getSiblingDB("nhs")
const clusterName = 'Cluster0' // Your is probably Cluster0 or
AtlasCluster
const projectId = '625eaf349bc6263074a36221'
const atlasDatabase = 'nhs'
const atlasCollection = 'prescriptions'

const outToAtlas = { $out: { atlas: { projectId, clusterName,
                                      db: atlasDatabase,
                                      coll: atlasCollection } } }
console.time()
db.cleanrawdata.aggregate([outToAtlas])
console.timeEnd()
```

We can now test a simple aggregation: calculating the total amount of spend on all of these prescriptions by summing the quantity multiplied by the per-item price in each record.

Note that when you create or replace a collection using $out in Atlas, the new version may not be visible to Federation until you log out and back in again.

```
const prescriptionPrice = { $multiply : [ "$items", "$actcost"]}

const totalCost = { $sum : prescriptionPrice }
const totalItems = { $sum: "$quantity" }
const nPrescriptions =  { $count : {} }

 /* The fields/expression to group by - or a constant  for one group */

_id = "allscripts" ;
const groupAll = { $group: { _id, totalCost,totalItems,nPrescriptions} }
console.time()
db.prescriptions_atlas.aggregate([groupAll])
console.timeEnd()
console.log("")
```

```
ADF> db.prescriptions_atlas.aggregate([groupAll])

 {
    _id: 'allscripts',
    totalCost: 2690707041572.53,
    totalItems: 705457122172.05,
    nPrescriptions: 1000643517
  }
]
ADF> console.timeEnd()
%s: %s default 37:15.273 (m:ss.mmm)
```

Now, we want to write that data back out to S3 in an increasing number of files and then test the query performance against it. We are not reading it back from the Atlas cluster and writing it but copying from S3 to S3, as copying from Atlas ordered in chunks requires a disk sort and that uses more disk space than we have available in this case.

Important: We are adding a field here to allow us to divide our data into an ***exact*** *number of files for measuring purposes. You should not do this in production. If you want to partition your data into multiple files without an explicit partition field, use the maxFileSize set to an appropriate value to partition your data as required.*

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'prescriptions_lake'

const s3StoreName = 's3bucket'
const s3Path = 'prescriptions/lake/{partition int}/'

const dataSources = [ { storeName:s3StoreName, path: s3Path} ]

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 let virtualcollection = virtualdb.collections
                              .find(c => c.name ===
virtualCollectionName)
 if (virtualcollection) {
  virtualcollection.dataSources = dataSources
} else {
   virtualcollection = { name: virtualCollectionName, dataSources}
   virtualdb.collections.push(virtualcollection)
 }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

```
db=db.getSiblingDB("nhs")
nParts = [1,4,16,64,256,512,1024,2048,4096]
nParts = [512]
for(let p of nParts) {

const partition = {$toInt  : { $multiply: [ { $rand : {}} , p ]}}
const addPartitionField = { $set : { partition} }
const outfilename = {"$concat": ["prescriptions/lake/",
                                 {$toString:"$partition"},
```

```
                                    "/"]}

const writeCompressedBSON = {
 $out: {
   s3: {
     bucket: 'jpage-datalake',
     region: 'eu-west-1',
     filename: outfilename,
     format: { name: 'bson.gz', maxFileSize: '2TB' },
     errorMode: 'stop'
   }
 }
}


console.log(`Writing data to ${p} partition(s)`)
db.cleanrawdata.aggregate([addPartitionField,writeCompressedBSON])

console.log(`Writing Done - testing query speed`)
const prescriptionPrice = { $multiply : [ "$items", "$actcost"]}

const totalCost = { $sum : prescriptionPrice }
const totalItems = { $sum: "$quantity" }
const nPrescriptions =  { $count : {} }


/* The fields/expression to group by - or a constant  for one group */
_id = "allscripts" ;
const groupAll = { $group: { _id, totalCost,totalItems,nPrescriptions} }
console.time()
console.log(db.prescriptions_lake.aggregate([groupAll]).next())
console.timeEnd()

}
```

```
Writing data to 1 partition(s) Done - testing query speed
{  nPrescriptions: 1000643517 }
53:23.646 (m:ss.mmm)

Writing data to 4 partition(s) Done - testing query speed
{  nPrescriptions: 1000643517 }
```

```
13:39.266 (m:ss.mmm)

Writing data to 16 partition(s) Done - testing query speed
{   nPrescriptions: 1000643517 }
10:12.560 (m:ss.mmm)

Writing data to 64 partition(s) Done - testing query speed
{   nPrescriptions: 1000643517 }
2:33.016 (m:ss.mmm)

Writing data to 256 partition(s) - Done - testing query speed
{   nPrescriptions: 1000643517 }
52.396s

Writing data to 512 partition(s)Done - testing query speed
{  nPrescriptions: 1000643517 }
37.927s

Writing data to 1024 partition(s) Done - testing query speed
{   nPrescriptions: 1000643517 }
26.395s

Writing data to 2048 partition(s) Done - testing query speed
{  nPrescriptions: 1000643517 }
24.756s

Writing data to 4096 partition(s) Done - testing query speed
{   nPrescriptions: 1000643517 }
25.702s
```

Atlas	1	4	16	64	256	512	1024	2048
37.15	53:23	13.39	10:12	2:33	52	38	26	25

Using Parquet for columnar queries and exporting

If we want to retrieve entire documents or rows for processing, then row-based formats, either tabular and untyped (like CSV or document-structured) and typed (like BSON) are useful. But when we want to focus only on a small subset of fields, it can be beneficial to use a columnar file format.

Columnar formats store all the values of a given field together, so instead of Columns A, B, and C for three records being stored in the file as…

A	B	C	A	B	C	A	B	C

… they are instead stored as:

A	A	A	B	B	B	C	C	C

This format means less CPU and file seeking is required to read all the values of a given field and also, if a given field has a pattern — such as many similar values or a slow increase — good compression algorithms can be applied. One of the most common columnar formats is Parquet. Parquet is also a common input format for other tools, and therefore, it is important we can output from MongoDB into Parquet to allow other tooling to use the data.

Using Data Federation, we can write out and query data in Parquet. Here we will compare the same data and query using compressed BSON and Parquet.

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'speedtest_bson'

const s3StoreName = 's3bucket'
const s3Path = 'prescriptions/speedtest_bson'
```

```
const dataSources = [
 { storeName:s3StoreName, path: s3Path},
 ]

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 let virtualcollection = virtualdb.collections.find(c => c.name ===
virtualCollectionName)
 if (virtualcollection) {
  virtualcollection.dataSources = dataSources
} else {
   virtualcollection = {
     name: virtualCollectionName,
     dataSources   }
   virtualdb.collections.push(virtualcollection)
 }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

```
db=db.getSiblingDB("nhs")

const outfilename = "prescriptions/speedtest_bson/data"


const writeCompressedBSON = {
 $out: {
   s3: {
     bucket: 'jpage-datalake',
     region: 'eu-west-1',
     filename: outfilename,
     format: { name: 'bson.gz', maxFileSize: '1GB' },
     errorMode: 'stop'
   }
 }
}



db.cleanrawdata.aggregate([writeCompressedBSON])

console.log(`Writing Done - testing query speed`)
```

```
const prescriptionPrice = { $multiply : [ "$items", "$actcost"]}

const totalCost = { $sum : prescriptionPrice }

_id = "allscripts" ; /* The fields/expression to group by - or a constant  for one group */
const groupAll = { $group: { _id, totalCost} }
console.time()
console.log( db.speedtest_bson.aggregate([groupAll]).next() )
console.timeEnd()

pAll]).next())
console.timeEnd()
```

```
const virtualdbName = 'nhs'
const virtualCollectionName = 'speedtest_parquet'

const s3StoreName = 's3bucket'
const s3Path = 'prescriptions/speedtest_parquet'

const dataSources = [
 { storeName:s3StoreName, path: s3Path},
 ]

const adminDB = db.getSiblingDB('admin')
const fedconf = adminDB.runCommand({ storageGetConfig: 1 }).storage

// Add collection if not there - virtual db already added

const virtualdb = fedconf.databases.find(db => db.name === virtualdbName)
if (virtualdb) {
 let virtualcollection = virtualdb.collections.find(c => c.name ===
virtualCollectionName)
 if (virtualcollection) {
  virtualcollection.dataSources = dataSources
} else {
   virtualcollection = {
     name: virtualCollectionName,
     dataSources   }
   virtualdb.collections.push(virtualcollection)
```

```
 }
}
console.log(fedconf)
adminDB.runCommand({ storageSetConfig: fedconf })
```

```
db=db.getSiblingDB("nhs")

const outfilename = "prescriptions/speedtest_parquet/data"


const writeParquet = {
 $out: {
   s3: {
     bucket: 'jpage-datalake',
     region: 'eu-west-1',
     filename: outfilename,
     format: { name: 'parquet', "maxFileSize" : "10GB", maxRowGroupSize: '1GB' },
     errorMode: 'stop'
   }
 }
}




db.cleanrawdata.aggregate([writeParquet])

console.log(`Writing Done - testing query speed`)
const prescriptionPrice = { $multiply : [ "$items", "$actcost"]}
const totalCost = { $sum : prescriptionPrice }
_id = "allscripts" ; /* The fields/expression to group by - or a constant  for one group */
const groupAll = { $group: { _id, totalCost} }

console.time()
console.log(db.speedtest_parquet.aggregate([groupAll]).next())
console.timeEnd()
```

	Time(s)	Num files (1GB each)	Rows
BSON.gz	246	44 x 1GB	1 Bn
parquet	28	12 x 700MB	1Bn

We can see the Parquet query is 10 times faster than BSON as we were able to read only a subset of the data. This makes Parquet an excellent format for tasks where you need to access and process just a single column or a couple of columns.

Our method of estimating query costs also overestimates with Parquet as the Federation engine is usually able to read only a subset of each partition, so any calculation from the prices function above should be treated as an upper bound on cost.

And finally...

We've covered a lot! You've learned how to configure and use Atlas Data federation both as cheap lukewarm data store, as federated access to multiple live MongODB clusters, as a high performance analytics environment and as an archiving capability . We hope you will revisit this book as needed to help you with tasks like this. If you made it to the end, thank you for reading. If you have questions or comments about anything you've learned, hit me up at https://www.mongodb.com/community/forums/.

Over time, I hope to add more tips and recipes, but for now, as the old cartoons say, "That's all, folks!"